TERMINATION OF LOGIC PROGRAMS: THE NEVER-ENDING STORY

DANNY DE SCHREYEAND STEFAAN DECORTE

▷ We survey termination analysis techniques for Logic Programs. We give an extensive introduction to the topic. We recall several motivations for the work, and point out the intuitions behind a number of LP-specific issues that turn up, such as: the study of different classes of programs and LP languages, of different classes of queries and of different selection rules, the difference between existential and universal termination, and the treatment of backward unification and local variables. Then, we turn to more technical aspects: the structure of the termination proofs, the selection of well-founded orderings, norms and level mappings, the inference of interargument relations, and special treatments proposed for dealing with mutual recursion. For each of these, we briefly sketch the main approaches presented in the literature, using a fixed example as a file rouge. We conclude with some comments on loop detection and cycle unification and state some open problems. ◁

1. INTRODUCTION

1.1. Motivation

The study of termination properties of Logic Programs is a fairly recently explored research topic. Before 1988, very few papers have been devoted to it. Since then, however, the activity in this area has strongly increased. We studied and collected in our reference list a total of 64 papers published on the topic over this period.

There are various motivations for the LP termination analysis work. The most important of them, especially in the early work, is related to *control generation* and *systematic program*

Work partly supported by Esprit Basic Research Project COMPULOG II, Project No. 6810.
Research associate of the Belgian National Fund for Scientific Research.
Supported by GOA, "Non-standard applications of abstract interpretation," Belgium.
Address correspondence to Danny De Schreye and Stefaan Decorte, Department of Computer Science, K. U. Leuven, Celestijnenlaan 200A, B-3001 Heverlee, Belgium, E-mail: {dannyd, stefaan}@cs.kuleuven.ac.be.
Received May 1993; accepted January 1994.

655 Avenue of the Americas, New York, NY 10010 0743-1066/94/$7.00

development. For many years, the ease with which the logic and the control components of a Logic Program can be dealt with as separate issues has been regarded as one of the main advantages of the paradigm. Building on the *Algorithm = Logic + Control* equation of [74], systematic LP program development can be addressed as a two phased process (see, e.g., [61]). First, provide a correct logic specification of the problem domain in terms of Horn clauses. Then, identify a suitable control for the obtained specification, and combine it with the specification to obtain a correct and efficient program.

A minimal requirement for a *suitable* control is that the program should terminate for any query of interest. As a result, termination analysis has been used to provide one of the basic sources of input for control generation. Another, equally important source of information is given by determinism analysis.

[86] has been a first major contribution along these lines and has inspired many other works on the topic. [107] is another milestone. Here, the same problem is formulated and solved in a deductive database context, where the necessity of executing declarative specifications is often unavoidable, and where the additional tradeoff between bottom up and top down evaluation strategies makes detection of (non)terminating subcomputations even more crucial.

In more recent work, additional motivations have been related to program verification, to the relevance of termination to LP approaches for nonmonotonic reasoning, to decidability issues, and to applications in abstract interpretation and program transformation.

The relevance of termination for program verification should be obvious. As an example, [98] presents an application proving the termination of a compiler.

Due to the treatment of negation as *finite failure*, termination proofs are also of great relevance to LP approaches to nonmonotonic reasoning. The notions of *acyclic* and *acceptable* programs, introduced in [7] and [11], provide characterizations of terminating normal programs. The papers provide proofs that the various semantics developed for normal programs coincide for these classes of programs.

Yet another motivation for termination analysis work is to obtain a better understanding of the decidability of the halting problem for subclasses of Logic Programs. Here, work has been done to identify the boundary between minimal subclasses of Logic Programs, which still have the expressibility of a Turing machine, and maximal classes for which the halting problem is decidable (see, e.g., [57]).

Finally, in recent work, termination analysis is being applied in abstract interpretation and transformation techniques. With respect to analysis, the *oracle semantics* of [16] forms the basis of a new class of abstract interpretation techniques, where termination analysis is incorporated to improve the precision of the inferred properties. On the level of transformation, [45] shows how termination conditions, adapted from termination analysis, can be used in the preconditions for applying the transformation rules in order to ensure preservation of finite failures.

1.2. An Initial Classification of Approaches

The different motivations for studying termination have been a first source of divergence in the development of different analysis techniques. For a termination analysis aimed at control generation, program verification, abstract interpretation, or transformation, *automation* is a major concern. This is not the case for approaches that aim to gain understanding on the semantics of normal programs or on the decidability issue. As a result, one basis for classification of the termination literature is to distinguish automatic approaches for

detecting (non)termination from more theoretical work, providing manually verifiable criteria for (non)termination.

To refine this classification, some comments on the undecidability of the termination problem are in order. Logic Programs can encode any computable function (see, e.g., [78]). Therefore, they inherit the undecidability of the halting problem for Turing machines. The undecidability means that no termination analysis technique can be encoded that is both *complete* (returns a "yes" answer for every terminating program and query) and *safe* (always terminates itself). Of course, the problem is semi-decidable. Given a program and query, mere execution is a complete, but unsafe, semi-deciding procedure. It should be clear that in termination analysis, one is more interested in safe, but incomplete, semi-deciding procedures.

In view of these observations, we can refine our classification of LP termination analysis works into three types of approaches:

1. Techniques that express necessary and sufficient conditions for termination. By undecidability, such conditions cannot be verified through any automatic tool. The purpose of these techniques is:
   — to provide a better understanding of the termination problem and of other problems that depend on it, such as the semantics of negation as finite failure,
   — to provide support for manual verification of termination properties,
   — to serve as theoretical frameworks on which automatic techniques can build. In particular, certain parts of such frameworks could involve decidable properties, and they could directly be integrated in an automatic approach, while other parts could be replaced by decidable, but weaker conditions.
   Typical examples of this line of work are [7, 11, 12, 10, 18, 19, 27, 28, 53, 110, 42].
2. Techniques that provide decidable *sufficient* conditions for the termination problem, or for some subproblem. As discussed above, the purpose of these techniques is to provide support for development, verification, analysis, and transformation tools. An additional purpose of many systems is to validate the practicality of the termination conditions formulated in techniques of the first type. We omit the list of references. In our full reference list, most works which are not cited under point 1 or 3 are of this type.
3. Techniques that prove decidability or undecidability for subclasses of programs and queries. A class of programs that has been studied in much detail are programs consisting of a single clause of the type $p(s_1, ..., s_n) \leftarrow p(t_1, ..., t_n)$, where $s_i, t_i, i = 1, ..., n$ are terms, and of one single fact. Such programs have been shown to be on the boundary of decidability. The computations they entail are often referred to as *cycle unification*. Some works here are [57–59, 51, 52, 20, 119]

From this classification, at least on the highest level, the main issues involved follow directly. For type 1) approaches, the main point is to come up with conditions that are easy to reason about for humans. Also, in view of providing support for automatic techniques, the conditions should be easily decomposable into some well-identified undecidable components and their automatable complement. Most often, the latter is achieved by requiring the existence of a well-founded order defined in terms of the program (and possibly the query). By imposing certain properties on this order, the equivalence with termination is established. Here, the selection of an appropriate order can be seen as the undecidable component. The verification of the properties imposed on the order is its decidable complement.

For type 2) approaches, in light of the discussion above, a main issue is to automatically

generate useful orders, capable of proving termination for a large class of programs. Another issue here is the efficiency of the analysis. This is particularly the case for techniques applied at run time, such as [107].

Other main issues are of a more technical nature. They are closely related to some specific features of Logic Programs and the problems these features cause in the analysis. Specifically, backpropagation of bindings during unification and the occurrence of local variables in bodies of clauses require special care. Both type 1) and type 2) approaches need to deal with these problems, although their solutions are on a different level. We clarify the problems in Section 1.4 and present various solutions in the remainder of the paper.

1.3. Defining the Termination Problem

An immediate observation that one can make when going through the literature on LP termination is that there exists a variety of different definitions of termination. This can be related to three specific LP features: the absence of directionality and type declarations, nondeterminism through backtracking, and, for some LP languages, the possibility of using different selection rules and other control constructs. We illustrate the divergence in terminology caused by these features using the least original of all programs: *append.*

Example 1.1. (append)

$\text{append}(Nil, x, x) \leftarrow$
$\text{append}([x|y], u, [x|z]) \leftarrow \text{append}(y, u, z)$

a) Using append as a procedure to concatenate two ground lists, e.g., with a query $\leftarrow$ append([1, 2], [3], u), results in a finite computation, with one successful derivation.
b) In verification mode, with all three arguments bound to ground lists, e.g., $\leftarrow$ append ([1, 2], [3], [4]), the program again terminates, either with one successful or with a failing derivation.
c) For a third directionality, where the program is used as a procedure to generate all pairs of lists (l_1, l_2), such that a ground list, l_3, in the third argument can be split up into l_1 and l_2, e.g., $\leftarrow$ append(l_1, l_2, [1, 2]), we again have termination, now with a finite number of successful derivations.
d) Activating the program with only its second argument bound to a ground list, l_2, with the aim of computing all pairs of lists (l_1, l_3), such that l_3 is the concatenation of l_1 and l_2, e.g., $\leftarrow$ append(l_1, [1, 2], l_3), produces an infinite sequence of successful derivations.
e) The same behavior is obtained for a query with all arguments free, $\leftarrow$ append(l_1, l_2, l_3).

By allowing partially instantiated structures in the query, we even obtain other behaviors.

f) For instance, with a query $\leftarrow$ append($[x|y]$, Nil, y), we get one single non-terminating derivation.
g) For $\leftarrow$ append($[x|y]$, y, $[z|y]$), we obtain one finite successful derivation, followed by an infinite one. □

1.3.1. Lack of Directionality and Types. Termination of Logic Programs cannot be studied independently from the query or set of queries of interest. This is quite different

from termination analysis in other contexts. For instance, a Term Rewrite System is called terminating if rewriting terminates for all terms. Also, for (moded and typed) imperative programs, the set of queries of interest needs no explicit specification: the calls of interest are those that correspond to the directionality and types of the top-level procedure. In LP, this set is not directly deducible from the program at hand, and either it needs to be explicitly specified as an input to the termination analysis, or the termination analysis should be designed to infer a set of terminating queries. This has been a source of divergence in the LP termination literature: different techniques impose different limitations on the sets of queries that can be specified or inferred. We briefly list some representative choices that have been made:

— the study of termination for all *ground queries.*
— the specification of queries of interest through *modes.*
— generalizations of the above, specifying queries that are *bounded* or *rigid* with respect to a measure function (see Sections 3.1 and 3.2 for formal definitions). The aim of these generalizations is to allow specifications of queries involving partially instantiated data.
— sets of queries specified through combinations of modes and types. Again, these are aimed to allow the specification of partially instantiated data.

1.3.2. NONDETERMINISM. Cases d), e), and g) of Example 1.1 illustrate another specific aspect of LP termination. Due to the inherent nondeterminism of the paradigm, it is not clear that we should consider these examples as nonterminating, as we did in Example 1.1. In all three cases, Prolog will produce an answer after a finite computation and return the control to the user, who can then decide whether or not alternative answers are required. On the level of termination analysis, this leaves us with the choice of studying either *universal termination* or *existential termination.* A program is called *universally terminating* for a given set of queries if, for all considered queries, the computation produces *all* solutions and then terminates. It is called *existentially terminating* if, for all considered queries, it either finitely fails or returns at least one solution (after which it may or may not enter an infinite computation). Clearly, universal termination implies existential termination. In Example 1.1, cases a), b), and c) are universally terminating. In addition, d), e), and g) are existentially terminating under the Prolog evaluation strategy.

1.3.3. AVAILABILITY OF DIFFERENT CONTROL STRUCTURES. Of course, Logic Programming represents not one single programming language, but a family of languages, each with its own operational semantics. This is a further cause of divergence in the literature. Even when we restrict the attention to pure normal programs, with SLDNF as the basis for the operational semantics, a wide range of actual procedures results from different choices for the *selection rule* (subgoal selection) and *search rule* (clause selection).

The importance of the latter is illustrated in Example 1.1g). For this goal, $\leftarrow$ append($[x|y]$, y, $[z|y]$), existential termination of the append program holds under the top-to-bottom search rule of Prolog. But the existential termination property is lost if we consider a search rule which selects clauses in the reversed order. Note, however, that in the context of universal termination, the search rule is irrelevant. Here, all derivations under the given selection rule must terminate, which is independent of the order in which clauses are selected. Throughout the remainder of the paper, unless explicitly stated otherwise, we restrict our attention to *universal termination* and we simply refer to it as termination.

The importance of the selection rule is illustrated with the permute program.

Example 1.2. (permute)

permute(Nil, Nil) ←
permute($[x|y]$, $[u|v]$) ← delete(u, $[x|y]$, w), permute(w, v)
delete(x, $[x|y]$, y) ←
delete(u, $[x|y]$, $[x|z]$) ← delete(u, y, z)

Under the left-to-right selection rule of Prolog, the query ← permute([1], y) terminates. We provide some explicit proofs for this statement in the following sections. With the opposite, right-to-left selection rule, we get nontermination: a repeated selection of the second clause for permute produces an infinite derivation. □

Since the left-to-right selection rule is standard in Prolog, it is not surprising that this rule has obtained most of the attention in the literature. Following [11], we refer to termination under the left-to-right rule as *left-termination.* [107] studies termination under the (non-standard) selection rule generated in the NAIL!-system. Also [88] and [103] study non-standard selection rules. Another type of analysis that received considerable attention studies termination of the program and given queries *under all selection rules.* Here, all derivations, for all considered queries and for all possible selection rules, are required to be finite.

Several papers study termination for other operational semantics than those that are purely based on SLDNF. Some work is devoted to termination of full Prolog [10, 18]. [93] and [97] analyze termination for Guarded Horn Clause-Programs. [84] addresses the termination problem for CLP languages.

To conclude the subsection, it should be noted that in addition to the three mentioned sources of divergence, most techniques are developed to cope only with certain subclasses of Logic Programs. In particular most works are, either implicitly or explicitly, restricted to definite programs. [11, 12, 18, and 116] are some notable exceptions that explicitly address negation. Another restriction which is frequently imposed is the exclusion of indirect or mutual recursion. In any case, with the exception of [10] and [18], all works exclude the nonpure features of Prolog.

1.4. Two Technical Problems in LP Termination Analysis

The issues raised in previous subsections merely present different alternatives for *defining* the notion of LP termination. As yet, no mention is made of aspects involved in solving the resulting termination problem. In Section 1.2, we announced that backpropagation of bindings and the occurrence of local variables are two LP features that require *specific techniques* in LP termination analysis. In this section, we introduce the problems caused by these features and indicate some directions taken to solve them. A more technical account of these solutions is included in the following sections.

Frequently, we will relate the issues to termination analysis in Term Rewriting. This comparison is useful since the topic has received a great deal of attention in this field (see, e.g., [56]). First, we briefly point out the basic reasoning underlying termination proofs in general.

1.4.1. WELL-FOUNDED ORDERS AND TERMINATION PROOFS. A commonly used approach for proving termination of a computation process is to reason about the trace of the computation. More precisely, one proves that this trace can be ordered by means of a *well-founded ordering*. The ordering should be such that, between every two consecutive states in the trace, the order decreases. By well-foundedness of the ordering, this implies that the trace cannot contain an infinite sequence of consecutive states. Thus, the computation terminates.

Well-foundedness has been used for proving termination in many different contexts: for imperative programs [64], for production systems [80], and Term Rewrite Systems (see, e.g., [56]). It also forms the basis of most approaches to termination analysis in LP.

Above, the notion of a *trace* should be interpreted broadly. For Logic Programs, it could stand for SLD-tree (or, collection of SLD-trees, in the context of SLDNF), proof-tree, AND-OR-tree, and many others. In general, any representation of the computation will do, as long as finite parts of it cannot be the representation of infinite processes in the actual computation.

Of course, in the termination proof, one does not want to construct the trace explicitly and then verify well-foundedness under some ordering since this construction itself may be trapped in nontermination. Furthermore, verification proofs in general can usually be strongly simplified by abstracting away from computations. Thus, one aims to formulate and verify conditions in terms of some finite syntactic structures that imply well-foundedness of the trace.

For instance, in the context of Term Rewriting, the trace could be the union of all rewrite trees for all possible terms. The syntactic structures are precisely the rules of the given rewrite system. The condition to verify is that there exists a well-founded ordering on terms, such that for every instance (substituting variables by terms) of each rule, the term on the right-hand side of the instance is smaller than the term at the left-hand side. In addition, the ordering must have the *replacement property*, which we will not define here since it has no counterpart in LP termination.[1]

1.4.2. BACKPROPAGATION OF BINDINGS. In an attempt to reformulate these conditions for Logic Programs, the natural adaptation is to replace "term(s)" by "atom(s)" since atoms form the basic unit of any representation of a computation state in LP. However, a direct reformulation along these lines does not result in a sufficient condition for termination. Consider again the recursive clause for append:

$$\text{append}([x|y], u, [x|z]) \leftarrow \text{append}(y, u, z)$$

We can define a well-founded, strict partial order on append-atoms by $\text{append}([t_1|t_2], t_3, t_4) > \text{append}(t_2, t_5, t_6)$, for all terms $t_i, i = 1, 6$. With the given ordering, for any instance of the clause, the atom on the right-hand side is smaller than that on the left-hand side. Still, for some queries, for example, Example 1.1d), the program does not terminate.

The cause of this difference with Term Rewrite Systems is the use of unification instead of matching. Using matching, the application of a rule to a term has no effect on the term itself. With unification, performing a derivation step for a goal with free variables, e.g., $\leftarrow \text{append}(x, [1, 2], y)$, may instantiate these variables. As a result, on the level of the trace, we may end up with an infinite derivation in which the root (and, in fact, all other states) contains infinite terms.

[1] For the interested reader, the replacement property is needed in Term Rewriting because rules can be applied to rewrite a *subterm* of a given term. In Logic Programs, rules can only be applied to entire atoms.

Of course, the problem is very closely related to the lack of directionality, discussed in Section 1.3.1. But, in the context of formulating and verifying termination conditions, there is one important new element. Not only do we need a specification of the set of queries of interest, we also need specifications of the sets of calls to other predicates that can occur in any derivation. Using these specifications, we can verify whether or not unification is causing new instantiations in the call that change the order of the atom in the considered well-founded ordering. As an example, in Example 1.1a), a simple mode analysis reveals that *any* call to append in any derivation will have its first argument ground. Thus, unification of such a call with the head of the recursive clause has no effect on the order of the call and the program terminates.

Considering only those programs and queries for which each call in every derivation contains only *ground* input is a solution that has been taken by many LP termination techniques in the past (e.g., [107, 91, 96]). In the last few years, however, much attention has been devoted to extending the techniques to the nonground case. Similar to what was mentioned in Section 1.3.1, the notions of boundedness [19], rigidity [27], and combinations of modes and types [111, 33] have been designed for this purpose.

1.4.3. LOCAL VARIABLES. In a Horn clause, the body of the clause may contain variables that are not present in the head. One usually refers to them as *local variables*. This generality is not allowed in Term Rewrite rules. It is the cause of what is probably the most important technical difference between termination analysis in the two fields.

Consider again the recursive clause for permute from Example 1.2:

$$\text{permute}([x|y], [u|v]) \leftarrow \text{delete}(u, [x|y], w), \text{permute}(w, v)$$

and assume that we aim to prove left-termination for the query $\leftarrow \text{permute}([1], x)$.

For the query of interest, the input arguments of all descending calls will be ground, so that the backpropagation problem does not arise. The difficulty in this example is that it is impossible to provide a well-founded ordering on atoms, such that for each instance of the clause, the order of the head is larger than that of the atoms in the body. More specifically, focusing on the two permute atoms, we would like the order to be based on properties of the first argument of permute since this is the input argument. But these arguments, $[x|y]$ and w, have no syntactic relation (w is a local variable), and therefore we fail to order all their instances.

The solution is that, since we are proving left-termination, we do not need to show a decrease in order between the two atoms for *every instance* of the clause. Under the left-to-right selection rule, the recursive call to permute will only be selected after successful termination of the intermediate call to delete. As a result, we only have to provide an ordering which has a decrease between the two permute atoms, for those instances of the clause with a delete atom which is a computed answer for delete. Only those instances can occur under the given selection rule.

In the example, a useful order on permute atoms is: $\text{permute}(s_1, s_2) < \text{permute}(t_1, t_2)$ whenever s_1 and t_1 are lists and the length of s_1 (counting the length of the tail of an open-ended list as zero) is strictly smaller than that of t_1. Given any instance

$$\text{permute}([x_1|y_1], [u_1|v_1]) \leftarrow \text{delete}(u_1, [x_1|y_1], w_1), \text{permute}(w_1, v_1)$$

such that $\text{delete}(u_1, [x_1|y_1], w_1)$ is a computed answer for some query to delete, it can be proven that

1. $[x_1|y_1]$ and w_1 are lists, and

2. the length of $[x_1|y_1]$ is strictly larger than that of w_1.

Thus, the order decreases between the two permute atoms in such clause instances, establishing termination for the given query.

The example shows that we need techniques to prove properties of computed answers for certain atoms.

This observation has had two major impacts on automatic LP termination techniques. First, we need techniques to infer relations that hold between the terms in the computed answer instances of a given atom. In the example: the properties 1) and 2) above. Such relations are referred to as *interargument relations*. These techniques are surveyed in Section 4.2. Second, orders on atoms are most often defined in terms of orders on certain terms that occur within these atoms. In the example: the list-length of the term at the first argument position of the permute atom. Such orders on terms are referred to as *norms*. The restriction to norms has been useful to allow the development of computationally feasible techniques to infer interargument relations. We survey the study of norms in Section 4.1.

1.5. Plan of the Paper

The remainder of the paper is organized as follows. The next section introduces some conventions, notations, and basic definitions. In Section 3, we present and discuss some frameworks for LP termination. Here, the emphasis is on the global structure and strategy of the termination proof. In Section 4, we focus on more technical and fine-grained subtasks that are needed within these strategies. We classify them as: issues related to the well-founded measures and norms, inference of interargument relations, and dealing with mutual recursion. We end with a brief discussion of contributions that fall outside the scope of the general outline of the paper, and we present some open problems.

2. PRELIMINARIES

2.1. Notation

We assume familiarity with the terminology, the basic concepts, and results of Logic Programming, as they are, for instance, presented in [78] and [6].

We use the following notational conventions. Variables, functors, and predicate symbols start with a lower case character. Constants start with an upper case character. The Prolog notation p/n is used to represent a predicate with symbol p and arity n. If there is no risk of confusion, we simply write p. We also use the Prolog conventions for representing lists. Substitutions are denoted by Greek characters. N-tuples of indexed objects, e.g., variables or terms $(x_1, x_2, ..., x_n)$, are denoted as $\bar{x}$.

Given a program P based on a first order language $\mathcal{L}$, we denote its Herbrand Base as B_P. The set of all terms and the set of all atoms based on $\mathcal{L}$ are, respectively, denoted as $Term_P$ and $Atom_P$. For any expression E (a term, atom, n-tuple of terms, or n-tuple of atoms), $Var(E)$ denotes the set of variables occurring in E. If E and F are two expressions of the same type and the expressions unify, then mgu(E, F) denotes their most general unifier. The variant relation, both over $Term_P$ and over $Atom_P$, is denoted as $\sim$. We use $Term_P/\sim$ and $Atom_P/\sim$ to represent, respectively, the sets of equivalence classes of $Term_P$ and $Atom_P$ modulo $\sim$. $\models$ denotes logical consequence in the first order language.

From this section onwards, we denote a query as the sequence of its literals, separating literals by commas. This is convenient since it allows us to view sets of atomic queries

as sets of atoms. We take the convention that a program *does not* include a query. A pair (P, Q) consisting of a program P and a query Q is referred to as a *queried program.*

2.2. Termination

Restricting our attention to universal termination, the following generic definition of termination is useful in light of our discussion in Section 1.3.

Definition 2.1. (termination, generic)
Let P be a program, S a set of queries, and $\mathcal{R}$ a set of selection rules. P is *terminating with respect to S and $\mathcal{R}$* if, for each query Q in S and for each selection rule R in $\mathcal{R}$: all SLDNF-trees for the queried program (P, Q) under the selection rule R are finite. □

This definition covers many notions of termination studied in the literature. In particular, for $\mathcal{R} = \{\text{LDNF}\}$, where LDNF is the left-to-right selection rule of Prolog, it defines left-termination for the queries in S. Taking $\mathcal{R}$ equal to all selection rules yields another instance discussed in Section 1.3.3. The various choices that can be made on the level of the query-specification—see Section 1.3.1—are parameterized through S.

Throughout the remainder of the paper, we only consider termination analysis for *atomic* top-level queries. Although several works are formulated in terms of general top-level queries, the restriction to atomic ones does not cause any loss of generality. Given a program P and a set of compound top-level queries of interest, $\{p(\bar{x}), q(\bar{y}), ..., r(\bar{z}) | S(\bar{x}, \bar{y}, ..., \bar{z})\}$, where $S(\bar{x}, \bar{y}, ..., \bar{z})$ is some formal specification of the arguments $\bar{x}, \bar{y}, ..., \bar{z}$, under consideration, the termination problem is equivalent to that of $\{s(\bar{x}, \bar{y}, ..., \bar{z}) | S(\bar{x}, \bar{y}, ..., \bar{z})\}$, for the program $P' = P \cup \{s(\bar{x}, \bar{y}, ..., \bar{z}) \leftarrow p(\bar{x}), q(\bar{y}), ..., r(\bar{z})\}$, where s is a fresh predicate symbol.

2.3. Well-Moded and Correctly Asserted Queried Programs

The main part of this section is devoted to the definitions of *well-moded* and *correctly asserted* queried programs. These concepts turn up in several techniques. Their purpose is to facilitate formal reasoning on certain properties of a queried program's execution under some selection rule. Here, the properties in focus are modes and generalizations of modes, adequate for reasoning about computations with partially instantiated data. As such, their use in termination analysis is related to the backpropagation problem discussed in Section 1.4.2.

We first introduce well-moded queried programs.

Definition 2.2. (mode)
A mode for a predicate p/n is a function $m_p : \{1, 2, ..., n\} \rightarrow \{in, out\}$. If $m_p(i) = in$, we say that i is an input argument of p, else i is an output argument. A queried program (P, Q) is *moded* if it is augmented with a mode for every predicate in (P, Q). □

Modes have been studied first by C. Mellish in [83]. Since then, numerous works have been devoted to it. For notational convenience, we represent a mode m as $p(m_p(1), ..., m_p(n))$. As an example: consider permute(in, out). It is also notationally convenient to assume that the argument positions of each predicate in a moded queried program are reordered such that all input arguments precede all output arguments. This allows us to represent any atom in the moded queried program as $p(\bar{s}, \bar{t})$, where $\bar{s}$ is the m-tuple of all

terms on input argument positions of p/n and $\bar{t}$ is the $(n - m)$-tuple of all its terms on output argument positions.

The following definition implicitly assumes the use of the left-to-right selection rule.

Definition 2.3. (well-moded queried program)
Let (P, Q) be a moded queried program, with both P and Q definite.

The query $Q = p_1(\bar{s}_1, \bar{t}_1), ..., p_n(\bar{s}_n, \bar{t}_n)$ is *well-moded*, if
$Var(\bar{s}_i) \subseteq \cup_{j=1}^{i-1} Var(\bar{t}_j)$, for all $i = 1, ..., n$

A clause $p_0(\bar{t}_0, \bar{s}_{n+1}) \leftarrow p_1(\bar{s}_1, \bar{t}_1), ..., p_n(\bar{s}_n, \bar{t}_n)$ of P is *well-moded*, if
$Var(\bar{s}_i) \subseteq \cup_{j=0}^{i-1} Var(\bar{t}_j)$, for all $i = 1, ..., n + 1$

The queried program (P, Q) is *well-moded* if both Q and all clauses of P are well-moded. □

Example 2.1. (permute)
For the permute program and a query Q = permute(x, y), consider the modes: permute(in, out) and delete(out, in, out). A trivial transformation, switching the first two arguments of any delete atom in the program, makes this conform with our notations in Definition 3, with the adapted mode delete(in, out, out). One easily verifies that the queried program is well-moded. The program is also well-moded for the modes permute(in, in) and delete(in, in, out). □

Intuitively, a moded queried program is well-moded if

1. input variables of a body atom occur either as input variables in the head or as output variables of a body atom to the left of the atom,
2. output variables in a head atom occur either as input variables of the head or as output variables of any body atom.

As a result, under the left-to-right selection rule, well-moded queried programs are *data-driven*. This means that in any derivation under the selection rule, all input arguments of any selected atom are ground. From this observation, the usefulness of this notion in the context of the backpropagation problem is obvious: on input argument positions, no backpropagation can occur.

Well-moded programs were first introduced and studied by Dembinksi and Maluszynski in [55]. We used a formulation due to Rosenblueth [99].

The notion can straightforwardly be generalized to other selection rules, as long as the selection is based on a local order among body atoms of each clause. It has also been generalized to normal programs in [5].

[96] and related works use a more general notion of well-modedness. The main difference is that the concept is introduced independently of a given selection rule and query. We recall the following definitions from [96].

Definition 2.4. (producer/consumer)
Let C be a clause in a moded program. An atom A of C is a *producer*, respectively *consumer*, of a variable, x, if either

- A is the head of C and x is in an input (respectively, output) position of A, or,
- A is a body atom of C and x is in an output (respectively, input) position of A. □

Definition 2.5. (producer–consumer relation)
The *producer–consumer relation* of a clause, $C : A \leftarrow B_1, ..., B_n$, of a moded program is the relation
$\{(B_i, B_j) \mid$ there exists a variable x in C, such that B_i is a producer and B_j is a consumer of x $\}$. □

Definition 2.6. (well-modedness*)
A clause or a query, C, is *well-moded** if
1. every variable in C has a producer, and
2. the producer–consumer relation of C is acyclic.

A moded program is *well-moded** if all its clauses are *well-moded**. □

In [96], it is assumed that the considered selection rules induce a partial order on the body atoms of each clause in the programs, corresponding to the order of selection.
The following definition links the two notions.

Definition 2.7. (implied selection rule)
A selection rule, R, is *implied by* the modes of a well-moded* program, P, if for each clause C of P, the partial order induced by R on C contains the producer–consumer relation for C. □

Example 2.2. (permute)
The permute program is well-moded* for both the modes permute(in, out), delete(out, in, out) and permute(in, in), delete(in, in, out) of Example 2.1. The left-to-right selection rule is implied. It is also well-moded* for the modes permute(out, in), delete(in, out, in). Notice that the program augmented with the latter modes (and with any query) is not well-moded in the sense of Definition 3 due to the restriction to the left-to-right selection rule. However, the right-to-left selection rule is implied by the well-moded* program. □

While well-modedness is defined in terms of a given selection rule, well-modedness* allows us to determine a set of selection rules that are implied by it. As in the case of well-modedness, the main property of a well-moded* program is that it is data-driven for any well-moded* query and for any implied selection rule.

Although the condition of well-modedness is sufficient to avoid problems with back-propagation, due to the property of data-drivenness, it is not applicable in the study of termination of queried programs that compute on partially instantiated data. In this more general context, there is a need for similar, but generalized concepts. The notion of a *correctly asserted queried program* is one solution. It provides the basis of a very general method for reasoning about properties of a program's execution. Below, we recall the basic concepts involved: pre/post-specifications, asserted queried programs, and correctness of such programs. We refer to [26, 62], and [44] for more complete formal treatments. Again,

our definitions apply to the left-to-right selection rule only.

Definition 2.8. (pre/post-specification)
A *pre/post specification* for a predicate p/n in a program P is an expression of the form

$$\{Pre_p(\bar{x})\}p(\bar{x})\{Post_p(\bar{x})\}$$

where $Pre_p(\bar{x})$ and $Post_p(\bar{x})$ are formulas in a first order language $\mathcal{L}'$, whose set of functors includes the functors in the language $\mathcal{L}$ underlying P and whose set of constants includes the variables of $\mathcal{L}$. $\bar{x}$ is an n-tuple of variables in $\mathcal{L}'$. □

Definition 2.9. (asserted program)
An *asserted program* P is a program such that all predicates p of P have a pre/post-specification associated with them. □

Definition 2.10. (substitution closed formula)
A formula $F(\bar{x})$ in $\mathcal{L}'$, containing an n-tuple, $\bar{x}$, of free variables, is *substitution closed with respect to a theory* T associated to $\mathcal{L}'$, if for all n-tuples of terms, $\bar{t} \in Term_P^n$, and for all substitutions θ in $\mathcal{L}$:

$$T \models \forall(F(\bar{t}) \Rightarrow F(\bar{t}\theta)) \quad \square$$

Definition 2.11. (asserted queried program)
An *asserted queried program* P is an asserted program augmented with a query $q(\bar{x})$ and a formula $Desc_q(\bar{x})$ in $\mathcal{L}'$, such that $Desc_q(\bar{x})$ as well as all pre- and post-specifications in P are substitution closed. $Desc_q(\bar{x})$ is referred to as the *call description* for $q(\bar{x})$. □

Finally, we need the following definition, which is an informal version of Definition 3.4 in [26].

Definition 2.12. (correct asserted queried program)
An asserted queried program P, with query and call description $(q(\bar{x}), Desc_q(\bar{x}))$, is *correct with respect to an associated theory* T, if for all left-to-right derivations for any goal $q(\bar{t})$, such that $T \models Desc_q(\bar{t})$,

1. for every selected atom, $p(\bar{s}) : T \models Pre_p(\bar{s})$
2. for every success instance of such an atom, $p(\bar{s}\theta) : T \models Post_p(\bar{s}\theta)$ □

Example 2.3. (data-driven moded queried program)
A simple and commonly used instance of these notions is the characterization of a data-driven queried program. Given a mode m_p for a predicate p/n, it can be represented using pre/post specifications as

$$\{\text{ground}(x_{i_1}), \ldots, \text{ground}(x_{i_m})\}\text{p}(\bar{x})\{\text{ground}(x_{i_1}), \ldots, \text{ground}(x_{i_n})\}$$

where $\{i_1, \ldots, i_m\} = \{i | m_p(i) = in\}$ and $\{i_{m+1}, \ldots, i_n\} = \{i | m_p(i) = out\}$. Here, ground/1 is defined in some associated theory T. This theory could be defined to contain a fact

$$\text{ground}(c)$$

for every constant c in $\mathcal{L}$, and a clause

$\forall \bar{x}(\text{ground}(f(\bar{x})) \Leftarrow \text{ground}(\bar{x}))$

for every functor f in $\mathcal{L}$.

If the moded queried program is data-driven, then it is a correct asserted queried program in the sense of Definition 12, with respect to the theory T. In [9], it is shown that well-moded queried programs are an instance of correctly asserted queried programs. It also provides other instances of the concept (e.g., well-typedness). □

Of course, correctly asserted queried programs aim to capture more general properties than ground dataflow. Properties relevant to termination analysis are related to invariance under substitution of certain arguments within a given order relation. We return to these notions in Section 3.2, where such properties are formally introduced.

3. FRAMEWORKS FOR REASONING ABOUT TERMINATION

In the Introduction, we intuitively sketched a possible structure of an LP termination proof. Here, we present in more detail several general structures for constructing such proofs. We first devote attention to the approach proposed in the works of Apt, Bezem, and Pedreschi [19, 7, 11, 12], which is widely considered as a main theoretical foundation for the topic. We then present the basics of two very similar variants of this approach, one inspired by [26], the other proposed in [53]. These variants adapt the former framework to a line of reasoning which is more often used in automatic systems (e.g., [107, 90, 116]). Next, we recall the key concepts of a totally different approach, introduced in [96], that transforms Logic Programs into Term Rewrite Systems and uses Term Rewriting termination techniques. Then, we outline the method of [100] as an example of a technique developed in a deductive database context. Finally, we briefly sketch some other frameworks, proposed in [110, 65, 14, 42], and [18].

3.1. Recurrency and Acceptability

We first present the approach of Apt, Bezem, and Pedreschi in the context of *definite programs*, thereby restricting our attention to [19] and [11]. We conclude the subsection with some comments on various extensions, including those to normal programs.

These works address termination with respect to the set of all ground atoms. [11] focuses on left-termination, while [19] studies termination under every selection rule. In both cases, a program P is said to be terminating (respectively, left terminating) if all SLD-trees for all ground atomic queries and all the considered selection rules are finite.

The key concept of [19] is *recurrency*. It is based on the notion of a *level mapping* (see also [40]).

Definition 3.1. (level mapping)
A *level mapping* is a function $f : B_P \rightarrow I\!N$. □

Definition 3.2. (recurrency)
A program P is *recurrent* if there exists a level mapping, f, such that for each ground instance $A \leftarrow B_1, ..., B_n$ of a clause in P:

$$f(A) > f(B_i), \qquad \text{for each } i = 1, \ldots, n. \quad \square$$

One of the main results of [19] is:

Theorem 3.1. A program P is recurrent, if and only if, it is terminating. □

We illustrate the use of the theorem with the delete example. First, we give a more formal definition of the list-length function, introduced in Section 1.4.3.

Definition 3.3. (list-length)
list-length: $Term_P \rightarrow \mathbb{N}$ is defined as

$$\begin{aligned} &\text{list-length}([t_1|t_2]) = 1 + \text{list-length}(t_2), && \text{with } t_1 \text{ and } t_2 \text{ any terms,} \\ &\text{list-length}(t) \quad = 0, && \text{otherwise} \quad \square \end{aligned}$$

Example 3.1. (delete, permute)
Let f be a level mapping defined on the delete program as

$$f(\text{delete}(t_1, t_2, t_3)) = \text{list-length}(t_2), \text{ for any ground } t_1, t_2, t_3 \in Term_{delete}$$

To prove recurrency, take any ground instance of the recursive clause for delete, say:

$$\text{delete}(p, [q|s], [q|t]) \leftarrow \text{delete}(p, s, t)$$

where p, q, s, t denote ground terms. We have

$$f(\text{delete}(p, [q|s], [q|t])) = \text{list-length}([q|s]) > \text{list-length}(s) = f(\text{delete}(p, s, t))$$

so that the program is recurrent. Therefore, all derivations starting in a ground atom $\text{delete}(t_1, t_2, t_3)$ are finite.
The permute program is not recurrent. Proving nontermination on the basis of Theorem 3.1 is often difficult since one then needs to prove that *no* level mapping can satisfy the recurrency condition. So, using a direct argument instead, consider the ground goal permute([1], [1, 2]). For this goal, there is a derivation in which delete(2, x, y) gets generated as a subgoal. Since the latter is clearly non-terminating, this derivation for permute([1], [1, 2]) is infinite. □

Note that, although the set of ground atoms is very important in the context of LP semantics, it is not particularly useful in programming practice. [19] extends Theorem 3.1 to deal with more practical sets as follows.

Definition 3.4. (boundedness)
Let f be a level mapping. An atom A is *bounded with respect to* f if the set $\{f(A\theta)|\theta$ a grounding substitution for $A\}$ is bounded in $\mathbb{N}$. □

Theorem 3.1 can be generalized as follows.

Theorem 3.2. If a program P is recurrent with respect to some level mapping f, then P terminates under every selection rule for all bounded atomic queries with respect to f. Conversely, if P terminates under every selection rule for all bounded atomic queries with respect to some level mapping, then P is recurrent. □

Notice that the theorem is not formulated as an "if and only if" statement. This is because in the conclusion of the second implication, P is not necessarily recurrent with respect to *the same* f.

Example 3.2. (delete)
With the level mapping of Example 3.1, an atom delete(t_1, t_2, t_3) is bounded if t_2 is a, not necessarily ground, list of fixed length. With the same argument as in Example 3.1, Theorem 3.2 allows us to conclude termination for all such atoms. With an alternative level mapping, $g(\text{delete}(t_1, t_2, t_3)) = \text{list-length}(t_3)$, for any ground $t_1, t_2, t_3 \in Term_{delete}$, delete is also recurrent. Again, using Theorem 3.2, we may now conclude that delete terminates for all atoms delete(t_1, t_2, t_3) such that t_3 is a list of fixed length. In fact, by defining the level mapping as the minimum of the two previous level mappings, we obtain termination for the union of the two corresponding sets of queries. □

In [11], the results of [19] are reformulated in the context of left-termination. Here, the basic concept replacing recurrency is *acceptability*.

Definition 3.5. (acceptability)
A program P is *acceptable* if there exist a level mapping, f, and a model, I, for P, such that for each ground instance $A \leftarrow B_1, ..., B_n$ of a clause in P:

$$f(A) > f(B_i), \text{ for each } i = 1, ..., n, \text{ such that } I \models B_j, \forall j = 1, ..., i-1. \quad \square$$

The definition expresses that, for ensuring left-termination, the level mapping only needs to decrease between the head of a ground instance of a clause and a corresponding body atom, B_i, provided that all atoms to the left of B_i already follow from the model. If one of these atoms to the left of B_i would not follow from the model, then SLD-resolution under the left-to-right selection rule would never reach the point in which B_i is selected. The refutation would fail before that. Thus, there is no reason to impose that the level mapping should also decrease for such atoms. This is completely in the spirit of the intuitions we presented in Section 1.4.3.

The following theorem rephrases Theorem 3.1 in the context of left-termination.

Theorem 3.3. A program P is acceptable, if and only if, it is left-terminating. □

For bounded goals, the analog of Theorem 3.2 holds (see [11]).

Example 3.3. (permute)
We show acceptability of permute. We extend the level mapping f of Example 3.1 on permute atoms of $B_{permute}$ by:

$$f(\text{permute}(t_1, t_2)) = \text{list-length}(t_1) + 1.$$

For the model I, let the domain of the interpretation be $\mathbb{N}$. The pre-interpretation is determined by the function $J : Term_{permute} \rightarrow \mathbb{N}$, $J(t) = \text{list-length}(t)$. More in particular, J maps every constant in $Term_{permute}$ to 0, the list-functor to the function $(x, y) \rightarrow 1 + y$, and every other functor in the language to the constant 0-function. Finally, I(permute) is the binary predicate on $\mathbb{N}$ which is true everywhere and I(delete) is the relation $\{(x, y, z) \in \mathbb{N}^3 | y = z + 1\}$. Observe that I is a model for the permute program.

For the permute predicate, this is obvious since I(permute) is true everywhere. In the case of delete, for all ground terms t_1, t_2, t_3 such that t_3 is the list obtained from the list t_2 by deleting its member t_1, we have that list-length(t_2) = list-length(t_3) + 1.
Since the clauses for delete have at most one body atom, acceptability of delete with respect to f and I coincides with recurrency of delete with respect to f, which we proved in Example 3.1.
Next, consider the recursive clause for permute. There are two inequalities that we need to prove for it. The first is that for any ground terms p, q, r, s, and t: f(permute([$p|q$], [$r|s$])) > f(delete(r, [$p|q$], t)). This reduces to list-length([$p|q$])+1 > list-length([$p|q$]), which is true.
The second inequality is that for any ground terms p, q, r, s, and t: f(permute([$p|q$], [$r|s$])) > f(permute(t, s)) should hold, given that $I \models$ delete (r, [$p|q$], t). This reduces to list-length ([$p|q$])+1 > list-length(t)+1, given that list-length([$p|q$]) = list-length(t)+1. Again, this is clearly the case, so that permute is acceptable and left-terminating. □

We briefly relate the approach to the general intuitions on termination analysis, presented in the Introduction. The approach deals with the problems caused by backpropagation (Section 1.4.2) by restricting the analysis to ground or bounded queries. It is not hard to prove that for a ground or bounded query, every descending atom in any derivation for a recurrent or acceptable program is also bounded. As a result, computations in which the level mapping of some atoms grow unboundedly, due to instantiations caused in a descending derivation, are avoided.

With respect to local variables, the model required in the definition of acceptability serves the purpose of an interargument relation: it provides information on the bindings that intermediate atoms in the body cause on the local variables in following atoms. Notice that the model used in Example 3.3, is in fact, the essence of the interargument relation for delete presented in Section 1.4.3.

The framework has been extended to normal programs. [7] investigates termination under all selection rules, while [12] covers left-termination.

[7] generalizes the results obtained for recurrent programs. First, the notion of a level mapping is generalized. Its domain is extended to the set of all literals by stating that $|\neg A| = |A|$, for all $A \in B_P$. The concept of an *acyclic program* is obtained by adapting Definition 2 to normal programs and to the extended definition of a level mapping. The following theorem is one of the main results of [7]:

Theorem 3.4. A normal program is terminating if it is acyclic. □

Just as for definite programs, the result extends to the termination of bounded queries. The converse of Theorem 3.4 does not hold. The cause is the possibility of *floundering* derivations. An SLDNF-derivation can terminate in one of *three* alternative states: either because it fails or succeeds, or because the last goal consists entirely of negative atoms containing variables. The derivation is said to have floundered. This form of termination is not captured by acyclicity, as the following example from [7] illustrates:

Example 3.4.

$$p(0) \leftarrow \neg p(x).$$

The program is not acyclic. $p(0) \leftarrow \neg p(0)$ is a ground instance of the clause, and, by definition, no level mapping can decrease between its head and body atom. However, the

program terminates. The only possible ground atomic queries, $p(0)$ and $\neg p(0)$, terminate because of floundering. □

An additional and closely related reason why the inverse implication does not hold is *safety*. We refer to [7] for an example. For nonfloundering programs, the notions of terminating and acyclic programs are equivalent.

Acyclic programs have several interesting properties that make them a valuable class to study. Without going into detail (we refer to [7] instead), we describe the most appealing ones. For acyclic programs, the various kinds of semantics defined for normal programs coincide. With respect to the declarative semantics, the 2- and 3-valued fixpoint semantics, which in general differ, now coincide. They also coincide with the perfect model approach and the completion semantics. Furthermore, under the assumption of nonfloundering, these semantics can also be better linked to the procedural semantics. More precisely, for acyclic programs, SLDNF-resolution is a *complete* proof procedure. Whether a ground (bounded) literal is a valid consequence of an acyclic program is a decidable property.

Another contribution of the paper is that a famous temporal reasoning problem, known as the *Yale Turkey Shooting problem* (see [68]), can elegantly be solved using an acyclic program. The problem has been used in [68] to illustrate that a number of well-known approaches for temporal reasoning have difficulties in correctly representing and solving this very simple application. [7] demonstrates that acyclic programs, using only negation as finite failure as a basis for nonmonotonic reasoning, are sufficient to deal with the problem, both on the knowledge representation and the temporal reasoning levels.

In [12], the framework is reformulated for the left-to-right selection rule. Similar to the specialization of recurrency to acceptability, the difference for normal programs between acyclicity and acceptability lies in the model, I, in the termination condition. The main new aspect involved for normal programs is that, in essence, I needs to be a model of the *completion* of the program. Before recalling the definition from [12], we need the following:

Definition 3.6. (depends on)

Let P be a normal program and p, q predicates. We say that p *refers to* q if there is a clause in P that uses p in its head and q in its body. The *depends on* relation is the reflexive, transitive closure of refers to. □

Let Neg_P denote the set of predicates in P which occur in a negative literal in a body of a clause from P and let Neg_P^* stand for the set of predicates in P on which the predicates in Neg_P depend. By P^-, we denote the set of clauses in P in whose head a predicate from Neg_P^* occurs.

Definition 3.7. (acceptability — normal programs)

Let P be a (normal) program, f a level mapping for P, and I a model of P whose restriction to the predicates from Neg_P^* is a model of $Comp(P^-)$. P is called *acceptable* with respect to f and I if for every ground instance $A \leftarrow L_1, ..., L_n$ of a clause in P,

$$f(A) > f(L_i), \text{ for each } i = 1, ..., n, \text{ such that } I \models L_j, \forall j = 1, ..., i-1. \quad \square$$

Again, acceptability is sufficient for left-termination and, restricted to nonfloundering programs, the notions are equivalent.

The problems related to floundering, both in the case of termination under any selection

rule and left-termination, are solved in [81]. The solution is based on replacing negation as finite failure by *constructive negation* (see [41]). A formal definition of an extension of SLD-resolution with constructive negation, SLDCNF-resolution, is given. The main results are that a program is respectively terminating and left-terminating with repect to SLDCNF-resolution and for all ground (or equivalently bounded) queries iff the program is, respectively, acyclic and acceptable.

[4] provides another extension of the framework. Here, the results on left-termination are generalized in the context of nonpure definite programs, including *builtins*, such as $var, nonvar, constant, compound, functor, arg, =, >, :=, \backslash ==$. The work studies *strong termination*, which is defined as left-termination for *all* queries. Due to the introduction of the nondeclarative features, this notion is not as restrictive as it would seem (see [4] for practical examples). To study the termination of such programs, a new semantics, called θ-semantics, is introduced. It generalizes the S-semantics of [63]. For a specific class of programs, called *homogeneous programs*, the notion of acceptability is generalized with respect to the θ-semantics. For such programs, these notions of acceptability and strong termination are shown to be equivalent.

In [13], two further extensions to the framework are provided. One is on modularization of termination proofs for definite programs. Several results are presented, both on the level of recurrency and of acceptability, which are in fact treated in a uniform way in this paper. Given two recurrent, respectively acceptable, programs P and Q, the results formulate conditions under which $P \cup Q$ is recurrent, respectively acceptable. Different results deal with different interdependencies between P and Q. Appropriate level mappings for $P \cup Q$ are defined in terms of the given level mappings for P and Q individually. [13] also adapts the framework with respect to the treatment of recursion. We refer to the next subsection for more details on the latter point.

3.2. *Tuning on Recursion*

For a Logic Program, nontermination can only be caused by infinite recursion. This is an important observation (see also [90]), since for, e.g., imperative languages, various iterative programming constructs—and goto statements—also need to be taken into account. Even for full Prolog, writing a nonrecursive, nonterminating program is extremely difficult and requires the use of *nonstandard* forms of assert and retract. This observation suggests that termination conditions can take more advantage of the recursive structure of the program than is the case for the techniques described in the previous subsection.

There are several issues involved in tuning on recursion. Some of them are specific for mutually recursive programs. In order not to mix all issues, we restrict our attention to direct recursive programs in this section, postponing issues specific to the treatment of mutual recursion to Section 4.3. Note, however, that most of the techniques discussed below were originally introduced in the context of mutual recursion, and that our reformulations for direct recursion should not be understood as limitations of the surveyed works.

For directly recursive programs, the intuition is that termination conditions only need to require a decrease, $f(A) > f(B)$, of the level mapping between the head, A, of a rule instance and every corresponding body atom, B, *having the same predicate symbol as* A. Unlike the recurrency and acceptability approach, most works (e.g., [53, 26, 27, 107, 91, 92, 116, 100], and [61]) propose termination conditions of this form, focusing on recursion.

To motivate the following definitions, note that, even in the context of directly recursive programs, imposing $f(A) > f(B)$ for atoms B with a different predicate symbol from A is certainly not redundant in the definitions of recurrency and acceptability. The point is

that groundness or boundedness is only given for the top-level query. Due to the condition $f(A) > f(B)$, the boundedness property is inherited by every other call in the derivations, solving the backpropagation problem.

As a result, termination conditions tuned on recursion need to compensate the omission of the decrease $f(A) > f(B)$ for body atoms, B, with different predicates from that of the head, A, by a condition imposing some form of boundedness on all descending calls.

One such condition is formulated in [13]. Here, the notions of *semi-recurrency* and *semi-acceptability* are introduced. They are shown to be equivalent to, respectively, recurrency and acceptability. In [13], the additional condition, mentioned above, is that the level mapping should *not increase* between the head and a non-recursive body atoms, of a ground instance of a clause. This condition still ensures boundedness of every atom in any derivation, but only imposes a well-founding on the recursive part of the derivations.

As a second example, we present a simplified version of the conditions introduced in [26]. The simplification is mostly due to our restriction to direct recursion. Here, a solution to the backpropagation problem is formulated using program assertions (see Section 2). The approach is developed for left-termination only. The following definitions are inspired by [26].

Definition 3.8. (generalized level mapping)
A generalized level mapping for P is a function $f : Atom_P/_\sim \rightarrow I\!N$. □

With some abuse of notation, we will apply generalized level mappings directly on elements of $Atom_P$, omitting reference to the variant equivalence classes.

Definition 3.9. (rigidity wrt a generalized level mapping)
An atom $A \in Atom_P$ is *rigid wrt a generalized level mapping*, f, if for all substitutions $\theta : f(A\theta) = f(A)$. □

The following sufficient condition for left-termination is adapted from [26] (Proposition 4.9).

Proposition 3.1. Let P be a correct asserted queried directly recursive program with associated query, call description, and theory, respectively: $q(\bar{x})$, $Desc_q(\bar{x})$, and T. P is left-terminating for all queries $q(\bar{t})$ such that $T \models Desc_q(\bar{t})$ if there exists a generalized level mapping, f, for P, such that:

1. for every predicate $p(\bar{x})$ in P: if $T \models Pre_p(\bar{t})$, then $p(\bar{t})$ is rigid with respect to f.
2. for every directly recursive clause in P, $r : A \leftarrow B_1, ..., B_{i-1}, B_i, B_{i+1}, ..., B_n$, for every body-atom B_i of r, with the same predicate symbol as A, and for every instance $r\theta$ of r such that $(T \models \forall(Pre(A\theta) \wedge (\wedge_{j=1}^{i-1} Post(B_j\theta))))$, we have

$$f(A\theta) > f(B_i\theta). \quad \square$$

We illustrate the condition with the permute program.

Example 3.5. (permute)
Let f be the generalized level mapping defined as

f(permute(t_1, t_2)) =list-length(t_1)
f(delete(t_1, t_2, t_3)) =list-length(t_2)

Permute is the asserted queried program consisting of the clauses of Example 1.2, augmented

with

— the pre/post specifications:

{rigid(permute(x, y), f)} permute(x, y) {rigid(permute(x, y), f)},
{rigid(delete(x, y, z), f)} delete(x, y, z) {$list - length$(y)= $list - length$(z)+1, $rigid(delete$(x,y,z),f)}, where T is a theory in which rigid(x, y) holds whenever x is an atom which is rigid with respect to y, in which list-length is defined as in Definition 3 and = /2 as the equality over $\mathbb{N}$,

— the query and call description:

(permute(x, y),rigid(permute(x, y), f)).

By definition of f, rigid(permute(x, y), f) is equivalent to "the first argument of the permute atom is a list of fixed length." Thus, the proof of left-termination is for atoms of this type. Similarly, rigid(delete(x, y, z), f) is equivalent to "the second argument of the delete atom is a list of fixed length." One can verify that this asserted queried program is correct with respect to T, although a formal proof requires proof techniques based on e.g., the axiomatic semantics of [44]. Condition 1) in Proposition 3.1 is trivially fulfilled since every precondition for every predicate explicitly contains the condition of rigidity of the atoms with this predicate with respect to f. We only verify condition 2) for the recursive clause for permute. The recursive clause for delete can be dealt with in a similar manner. Given an instance

$$\text{permute}([p|q], [r|s]) \leftarrow \text{delete}(r, [p|q], t]), \text{permute}(t, s)$$

such that the pre-specification of the head and the post-specification of the delete atom hold, we must prove that $f(\text{permute}([p|q], [r|s])) > f(\text{permute}(t, s))$. By the post-specification of the delete atom, we know that list-length($[p|q]$) = list-length(t) + 1. Thus,

$$\begin{aligned} f(\text{permute}([p|q], [r|s])) &= \text{list-length}([p|q]) = \text{list-length}(t) + 1 \\ &> f(\text{permute}(t, s)). \end{aligned}$$

Thus, the condition is fulfilled and the program is left-terminating for all queries in the call description. □

Let us briefly compare Proposition 3.1 to the notion of acceptability and (the extension to bounded goals of) Theorem 3.3. None of the concepts involved in the notion of a correct asserted queried program, nor condition 1) in Proposition 3.1, has a direct counterpart in the approach of Section 3.1. Their only purpose is to ensure a form of boundedness, not only of the top-level query, but also of all descending selected subgoals. The notion of rigidity is more restrictive than boundedness. However, we should point out that the formulation in Proposition 3.1 is also more restrictive on this point than the conditions formulated in [26]. The same is true for the distinction between the necessary and sufficient condition in the case of acceptability and the sufficient condition in Proposition 3.1. [26] also provides a necessary and sufficient condition, which we omitted due to space restrictions.

The model I which is explicit in the definition of acceptability has a counterpart in the post-specifications for the intermediate body atoms in Proposition 3.1.

The main reason why we reformulated the termination conditions of [26] in the above form is that most automatic termination analysis techniques make use of conditions that are instances or close variants of Proposition 3.1. The works of Ullman and Van Gelder [107] and Plümer [91, 92, 90] are based on termination conditions that are similar to Proposition 3.1, but restrict attention to well-moded programs. Another approach using this type of termination condition, but phrased with the full generality of rigidity instead of (ground)

well-modedness, is [93]. [61] and [116], if restricted to the directly recursive case, are also similar within the context of well-modedness.

A closely related approach is formulated in [117]. The main differences are that it is not restricted to the left-to-right selection rule, and that, unlike the work of Bossi *et al.*, pre/post-specifications are associated to each *literal* in the program. Apart from a correctness condition analogous to the one in Definition 12, the fact that pre/post-specifications are expressed on a more local level requires a *safety* condition. For direct recursion, a condition very similar to Proposition 3.1 is obtained. We return to the technique in Section 4.3.

We conclude the subsection with a brief description of another closely related approach, proposed in [53]. In form, the termination conditions presented here are more similar to recurrency and acceptability, although they are still tuned on recursion. A main difference is that termination and left-termination are characterized with respect to any set S of atoms. The method requires information on how the queries in S propagate throughout the considered derivations.

Definition 3.10. (call set)

The *call set*, $Call(P, S, R)$, associated to a program P, a set of queries S, and a set of selection rules R, is the set of all $A \in Atom_P/\sim$, such that a representant of A is a selected atom in some derivation for a pair (P, Q), $Q \in S$, under some selection rule of R. □

[53] suggests the use of abstract data domains for the specification of S and abstract interpretation over these domains as a way to approximate $Call(P, S, R)$ by a superset. In this sense, the notions of S and $Call(P, S, R)$ are very closely related to the formulas $Desc_q(\bar{x})$ and $Pre_p(\bar{x})$ of [26]. Program assertions, with proofs of properties via axiomatic semantics, and abstract data domains, with inference of properties through abstract interpretation, are known to be two closely related approaches, often geared towards solving similar problems.

Definition 3.11. (recurrency wrt S)

Let S be a set of atoms and P a directly recursive program. *P is recurrent with respect to S* if there exists a generalized level mapping, f, such that

- for any representant A' of $A \in Call(P, S, R)$, where R contains all selection rules,
- for any clause $A'' \leftarrow B_1, ..., B_i, ..., B_n$, such that $\text{mgu}(A', A'') = \theta$ exists,
- for any atom B_i, $1 \leq i \leq n$, with the same predicate symbol as A'':

$f(A') > f(B_i\theta)$. □

Proposition 3.2. A directly recursive program P terminates for any query in a set S if and only if P is recurrent wrt S. □

Similarly, [53] contains a reformulation of acceptability. It can be obtained from Definition 11 by requiring that: 1) in addition to the existence of f, there exists a model, I, for P, 2) the considered call set is $Call(P, S, \{LD\})$ instead of $Call(P, S, R)$, where LD is the left-to-right selection rule, 3) $f(A') > f(B_i\theta)$ should only hold for those $B_i\theta$ such that $I \models B_j\theta$, $1 \leq j \leq i - 1$. We have:

Proposition 3.3. A directly recursive program P is left-terminating for all queries in a set S if and only if P is acceptable with respect to S. □

With respect to [26], apart from the distinction of using abstract interpretation instead of pre/post specifications, the main differences are that: 1) the decrease is between the previous call and the next call, instead of between the head and the body-atom, and 2) the fact that no rigidity conditions (or any related ones) are imposed. To illustrate the latter point, consider the following example.

Example 3.6. (append)
Take the append program and a query append(x, y, z), with all its arguments free. Clearly, the program is nonterminating for this query. The termination conditions in Proposition 3.1 fail to hold because there is no nontrivial generalized level mapping such that $Pre_{append}(x, y, z)$ is rigid with respect to it. Using Proposition 3.2 or 3.3, nontermination follows from the fact that for any generalized level mapping, f, $f(A') = f(B_1\theta)$, for every pair of descending calls A' and $B_1\theta$ to append/3. In fact, all such calls are variants. □

Although the absence of any requirement of boundedness or rigidity in Propositions 3.2 and 3.3 is theoretically intriguing, it has little or no practical impact. Experiments illustrate that, in those cases in which these termination conditions hold, the atoms in $Call(P, S, R)$ are, in fact, bounded or rigid with respect to the selected generalized level mapping. Conversely, it is the case that for rigid calls, the generalized level mapping has the same value on the call and the corresponding head of any clause. So, the conditions of Proposition 3.1 also imply a decrease between every two calls. Moreover, in the generalizations to mutually recursive programs, formulated in [53], explicit rigidity requirements do turn up.

The above approach has been the basis for a fully automatic termination analysis system, described in [111] and [114]. Several components of the system, including the inference of interargument relations [113] (see also Section 4.2), rely on the use of abstract interpretation. Termination conditions are formulated as a system of linear inequalities, the solvability of which is automatically verified using the constraint logic programming system Prolog III [43].

As a final comment on the relation between the frameworks proposed in this subsection and the previous one, we should point out that the termination conditions in Section 3.1 require no information on *properties of derivations* of the considered queried program. This is *not* the case for the two main approaches discussed in the current subsection. Properties such as rigidity in pre/post-specifications and knowledge on the call set are explicitly related to derivations. Global analysis techniques are needed to infer or verify them.

3.3. A Transformational Approach

Some termination analysis techniques make use of program transformation. A first motivation for introducing transformation in an approach is to simplify the given program and/or query. As a typical example, mutually recursive programs can often be transformed into equivalent directly recursive ones (see [90] and Section 4.3). Another type of transformation maps the given Logic Program to an associated program in a different language. Typical target languages are Term Rewrite Systems or Functional languages. Here, the point is that for the target language, rich theories and powerful tools for termination analysis may be available. This is especially the case for Term Rewrite Systems. If the transformation is such that the termination properties of the transformed program are inherited by the Logic Program, then LP termination can benefit indirectly from the results obtained for the target language.

A very successful approach of this type is presented in [96]. The approach is developed for well-moded*, definite programs. The target language is Term Rewrite Systems.

As can be expected, the core of the approach is the transformation scheme. Since Term Rewrite Systems do not allow local variables, the transformation must eliminate all such variables. This is done by introducing Skolem functions. More precisely, for each moded predicate p of arity n and having k output positions, k new function symbols, p^1, p^2, ..., p^k, are introduced. Each function p^i has arity $n - k$. Then, for each defining clause for p and each output position i, a rewrite rule for p^i is introduced. Intuitively, the rule expresses how the input arguments of p need to be rewritten to obtain an output for the ith output argument of p.

We illustrate the transformation with the permute program.

Example 3.7. (permute)
To clarify the transformation, we recall the clauses from Example 1.2:

(1) permute(Nil, Nil) ←
(2) permute([x|y], [u|v]) ← delete(u, [x|y], w), permute(w, v)
(3) delete(x, [x|y], y) ←
(4) delete(u, [x|y], [x|z]) ← delete(u, y, z)

Assume that the modes permute(in,out) and delete(out,in,out) are given. With respect to these modes, the transformation introduces three function symbols: permute^2, delete^1, and delete^3, all of arity 1.
First consider clause 3). It gives rise to one rewrite rule for delete^1 and one for delete^3:

$$\text{delete}^1([x|y]) \rightarrow x$$
$$\text{delete}^3([x|y]) \rightarrow y$$

Observe that the right-hand sides of these rules are precisely the output values for the considered argument positions. For clause 4), the associated rules are slightly more complex, due to the recursion. We get

$$\text{delete}^1([x|y]) \rightarrow \text{delete}^1(y)$$
$$\text{delete}^3([x|y]) \rightarrow [x \mid \text{delete}^3(y)]$$

Clause 1) trivially transforms to: permute$^2(Nil) \rightarrow Nil$. Finally, for clause 2), we need to deal with the local variable w. Since this variable is on the third argument position of the delete(u, [x|y], w) atom, we can represent it as delete^3([x|y]). For similar reasons, u can be represented as delete^1([x|y]) and v as permute^2(delete^3([x|y])). Thus, we get the rule:

$$\text{permute}^2([x|y]) \rightarrow [\text{delete}^1([x|y])|\text{permute}^2(\text{delete}^3([x|y]))]$$

Note that the resulting TRS is in no way semantically equivalent with the given Logic Program. For instance, it can be used to rewrite the term permute^2([1|[2|Nil]]) to the term [2|[2|Nil]]. □

An additional aspect of the transformation is that it may be necessary to introduce inverse functions for the functors occurring in the Logic Program. This is the case when a compound term occurs on an output position in the body of a clause. As an example, assume that the modes for the permute program are permute(out,in) and delete(in,out,in). Now, the generated functions are permute^1 of arity 1 and delete^2 of arity 2. Consider clause 2). The local variable w causes no new problems. It can be represented as permute$^1(v)$. However,

the output variables x and y in the second argument of the atom delete(u, $[x|y]$, w) cannot be directly represented in terms of the input variables and the available functors. We need to introduce two inverse functions for the [.|.] functor, say car and cdr, both of arity 1, which allow us to represent x as car(delete^2(u,permute^1(v))) and y as cdr(delete^2(u, permute^1(v))). The resulting TRS now is:

$$\begin{aligned}
&\text{permute}^1(Nil) &&\to Nil\\
&\text{permute}^1([u|v]) &&\to [car(\text{delete}^2(u, \text{permute}^1(v)))|\text{cdr}(\text{delete}^2(u, \text{permute}^1(v)))]\\
&\text{delete}^2(x, y) &&\to [x|y]\\
&\text{delete}^2(u, [x|z]) &&\to [x|\text{delete}^2(u, z)]\\
&\text{car}([x|y]) &&\to x\\
&\text{cdr}([x|y]) &&\to y
\end{aligned}$$

Note that in this specific example, the introduction of car and cdr could be avoided since neither x nor y occurs in isolation on an output position in a head. In fact, the transformation could easily be adapted to generate a more natural version of the second rule for permute^1, avoiding inverse functions. However, the example sufficiently illustrates the point. Furthermore, we use it to clarify some further issues below.

The main result of [96] is the following.

Theorem 3.5. A well-moded program, P, terminates for all well-moded* queries, Q, and under any selection rule implied by the modes (see Section 2), if its associated TRS terminates.* □

Example 3.8. (permute)
The TRS derived for the modes permute(in,out) and delete(out, in, out) in Example 3.7 is nonterminating. No conclusion on the termination of the well-moded* program can be drawn. In particular, left-termination of permute for queries permute(t_1, t_2), with t_1 ground, cannot be proved on the basis of Theorem 3.5. We comment on this example below. For the second well-moding*, permute(out, in) and delete(in, out, in), however, the resulting TRS terminates. Theorem 3.5 allows us to conclude termination for all queries permute(t_1, t_2), with t_2 ground, and for all selection rules implied by the modes. In particular, the queries terminate under the right-to-left selection rule. □

The example illustrates both advantages and drawbacks of the approach. First, it should be noted that a TRS is said to be terminating if the rewriting terminates *for any order of application* of the rules. This is the reason why, in the first part of our example, we were unable to prove left-termination. In this example, one possible rewrite order is to continuously alternate the application of the second rule for delete^3 and the second rule for permute^2, which does not terminate. In particular, an initial term permute^2($[A|Nil]$) is rewritten as:

$$\begin{aligned}
&[\text{delete}^1([A|Nil])|\text{permute}^2(\text{delete}^3([A|Nil]))]\\
&[\text{delete}^1([A|Nil])|\text{permute}^2([A|\text{delete}^3(Nil)])]\\
&[\text{delete}^1([A|Nil])|[\text{delete}^1([A|\text{delete}^3(Nil)]|\text{permute}^2(\text{delete}^3([A|\text{delete}^3(Nil)]))]]\\
&\ldots
\end{aligned}$$

Note that the corresponding query permute($[A|Nil]$, res) to the LP program has a similar infinite derivation under the corresponding selection rule. Here, the sequence of descending

goals is

delete(u_1, $[A|Nil]$, w_1), permute(w_1, v_1) $(res = [u_1|v_1])$
delete(u_1, Nil, z_2), permute($[A|z_2]$, v_1) $(w_1 = [A|z_2])$
delete(u_1, Nil, z_2), delete(u_3, $[A|z_2]$, w_3), permute (w_3, v_3) $(v_1 = [u_3|v_3])$
...

Although in some cases the lack of focus on an individual selection rule is a draw-back of the method, in others it is an advantage. The method is not tied down to one specific selection rule, which is the case for many other approaches. Given a well-moded* program, if the associated TRS terminates, the method allows us to suggest a set of selection rules under which the program terminates. This is particularly well suited for control generation.

Some drawbacks of the method, as it is presented in [96], are the restriction to well-moded* programs and the fact that, due to the transformation, the method provides little new insight in LP termination.

[103] provides solutions for both issues. It adapts the approach in two different ways. A first adapted version allows us to treat termination under all selection rules. This is essentially based on a more simple transformation, under which a clause $H \leftarrow B_1, ..., B_n$ gives rise to n rewrite rules $H \rightarrow B_1, ..., H \rightarrow B_n$. The work also devotes attention to termination under one specific selection rule. This is done by encoding the selection rule within the terms and rewrite rules. This part of the paper links the approach more closely to other works in the area. A second extension, in the same paper, is to remove the restriction of well-modedness*. Within the context of termination under all selection rules, notions of boundedness, similar to the one of Subsection 3.1, are introduced. One notion is defined on the atoms of the Logic Program and a related notion on the terms of the Rewrite System. Termination of the Logic Program for bounded queries is then characterized in terms of termination of the corresponding TRS and boundedness of its terms.

Notice that the approach does not require the inference of models or interargument relations to deal with local variables. This is important because several techniques for inferring interargument relations, for various technical reasons, impose restrictions on the given programs (see Section 4.2). Since such restrictions are not needed here, the technique can prove termination for some programs that are beyond the scope of some other techniques. Termination of the ProCos compiler [98] is an example.

Finally, it should be noted that the technique can benefit from the extensive amount of work done on Term Rewrite Systems termination. Many term orderings, in particular, the simplification orderings (see [56]), such as recursive path ordering, lexicographic path ordering, and elementary interpretations, are available and are implemented in rewrite theorem provers, such as RRL [71] and REVE [76]. The method has been automated using RRL.

In [2], the idea of adapting the technique to the study of termination under one specific selection rule is elaborated in much more detail. Here, the logic program, P, and query, Q, are transformed into a TRS, T, and a starting term, t, respectively. It is proved that if P is well-moded with respect to the left-to-right selection rule, then t has a finite reduction tree under T iff (P, Q) is left-terminating. This is achieved through a more specific transformation that *compiles* the selection rule.

3.4. *A Deductive Database Approach*

A large amount of work has been devoted to termination of query-evaluation in a deductive database context. Important contributions in this line of work are [1, 30, 31, 32, 73, 75,

95, 100, 101, 105, 107], and [109]. Distinction is made between the problem of proving termination under top-down evaluation and proving that a bottom-up evaluation generates a finite answer. The latter property is referred to as *safety*. In what follows we restrict attention to top-down evaluation, merely mentioning [73, 75, 95], and [101] as some approaches for studying safety.

Several of the contributions to top-down termination analysis are focused on the inference of interargument relations. We review these works in Section 4.2, which is entirely devoted to this problem. The main contribution in [105] is to provide support for the automatic generation of appropriate level mappings. It is discussed in Section 4.1, in the context of providing well-founded orderings.

On the level of the underlying structure of the termination proofs, these approaches are very similar to the ones discussed in Section 3.2. Well-modedness is most often required and termination conditions are tuned on recursion.

Technically, two specific aspects, different from the previous frameworks, usually arise. First, a deductive database is composed of two components: an extensional database (EDB), which consists of a set of database facts, and an intensional database (IDB), which consists of rules defining how additional relations can be computed from the EDB. In most approaches, the IDB is assumed to be a Datalog Program. This is not a restriction with respect to general Logic Programs since the latter can be transformed to Datalog Programs with an infinite EDB (see, e.g., [95]). Of course, this extends the notion of a Datalog program.

Example 3.9. (permute)
The permute program can be transformed to the Datalog Program:

$$
\begin{aligned}
\mathrm{perm}(s, s) &\leftarrow P_{nil}(s).\\
\mathrm{perm}(s, t) &\leftarrow P_{[.]}(s, x, y),\\
&\quad\ \mathrm{del}(u, s, w),\\
&\quad\ \mathrm{perm}(w, v),\\
&\quad\ P_{[.]}(t, u, v).\\
\mathrm{del}(x, s, y) &\leftarrow P_{[.]}(s, x, y).\\
\mathrm{del}(u, s, t) &\leftarrow P_{[.]}(s, x, y),\\
&\quad\ \mathrm{del}(u, y, z),\\
&\quad\ P_{[.]}(t, x, z).
\end{aligned}
$$

The program is augmented with the infinite EDB for the $P_{[.]}$ relation, which defines the *cons* constructor. It contains such tuples as: (Nil, Nil, Nil), $(a, 1, Nil)$, where a denotes the list [1], $(b, 2, a)$, where b denotes the list [2, 1], etc. The relation P_{nil} has only one tuple, $P_{nil}(Nil)$. □

A second technical difference is that most approaches in this line of work rely on dataflow graphs for representing and expressing solutions for various tasks involved in the termination analysis, e.g., the layout of the termination conditions, the inference of interargument relations, the propagation of modes for the moded top-level query.

By way of example, we give an outline of the approach presented in [100]. It is formulated for Datalog. Note that for Datalog Programs augmented with a finite EDB, termination is decidable. In particular, the conditions formulated in [100] are necessary and sufficient for termination. The considered selection rule is the left-to-right one, but any other selection rule can be dealt with by adapting the notion of an *augmented argument mapping* (see below). The queries of interest are specified through modes.

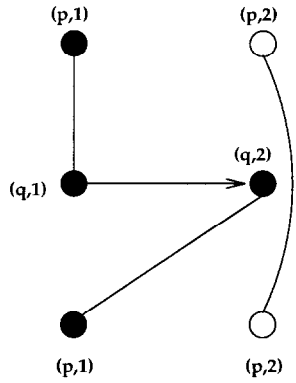

FIGURE 1. Example of an argument mapping.

The approach uses a well-founded ordering induced by so called *monotonicity constraints*. A monotonicity constraint for a predicate p has the form $p_j < p_i$, where p_i and p_j denote the sizes of the arguments on positions i and j. Several monotonicity constraints could be derived for one single predicate. [30] presents an algorithm for inferring them. In addition, in the context of programs with function symbols, or alternatively, their representations as Datalog Programs with an infinite EDB, a more general form of constraints, called *inequality constraints*, is needed. Their most simple form is $p_j \leq p_j + c$, where c is an integer. [32] proposes a technique for inferring them (see also Section 4.2).

Combinations of modes and monotonicity constraints, as well as the way in which they propagate through clauses, are represented by means of *argument mappings*. An argument mapping is a graph with two kinds of nodes, corresponding to either bound or free arguments. The nodes may be connected through either arcs or edges. An arc from a node (p, i) to a node (p, j) denotes a monotonicity constraint $p_j < p_i$. Edges represent occurrences of the *same* variable. An argument mapping can be constructed for each clause, making both modes and monotonicity constraints explicit. For instance, for the clause $p(x, y) \leftarrow q(x, z), p(z, y)$ with mode $p(in, out)$ and constraint $q_2 < q_1$, the associated argument mapping is presented in Figure 1.

An *augmented argument mapping* is an argument mapping associated to a rule and a selected atom from the body of the rule. It is organized in layers: one layer for each atom preceding the selected one. The top layer, corresponding to the head of the clause, is referred to as the *domain*, the bottom layer, corresponding to the selected atom, as the *range*. Equality information about the variables occurring in the clause, as well as the monotonicity constraints for all atoms *preceding the selected one*, are taken into account. A *summary* of a mapping restricts the graph to its domain and range and summarizes the connections between these nodes in a straightforward way. Figure 2 presents a summary for the augmented argument mapping of Figure 1, where the recursive call to p is the selected atom.

A *query pattern* is an argument mapping containing the argument positions of only one atom. A *query mapping pair* consists of a query pattern π and a summary mapping μ such that the domains of π and μ are labeled by the same predicate. Query mapping pairs (π, μ) may be *composed* with each other or may *generate* new pairs by coupling the range of the summary mapping with the summary of the composition of μ with any augmented argument mapping. If (π, μ) is a query mapping pair and the range of μ is identical to

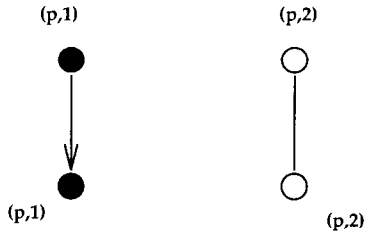

FIGURE 2. Summary of the argument mapping in Figure 1.

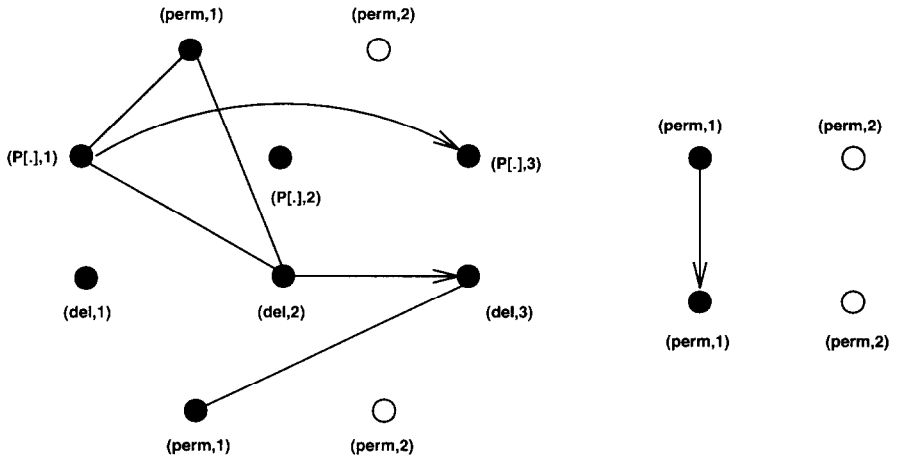

FIGURE 3. An augmented argument mapping and its summary for permute.

π, then we may construct a *circular variant* (π, μ') out of it by connecting corresponding nodes of domain and range by an edge. Given a top-level query pattern κ, the termination conditions are defined on the circular variants of the set $\Re(P, \kappa)$ (the set of *relevant* query mapping pairs), which is defined as the minimal set of query matching pairs containing the pair (κ, κ) which is closed under generation and composition.

The following theorem from [100] characterizes left-terminating Datalog Programs with respect to a given set of monotonicity constraints.

Theorem 3.6. A Datalog Program P is left-terminating for all queries matching a pattern κ *if and only if every circular variant of* $\Re(P, \kappa)$ *has an arc oriented from domain to range.* □

Example 3.10. (permute)
Assume that we aim to prove left-termination for the Datalog version of the permute program in Example 3.9, with respect to the mode $perm(in, out)$. We assume that the monotonicity constraints $del_3 < del_2$ and $P_{[.]_3} < P_{[.]_1}$ have been derived. First, we compute all augmented argument mappings. As an example, the one for the recursive clause for perm and the third atom in its body is represented in Figure 3.

Next, we compute $\Re(P, \kappa)$ by generating new query-mapping pairs and by composing

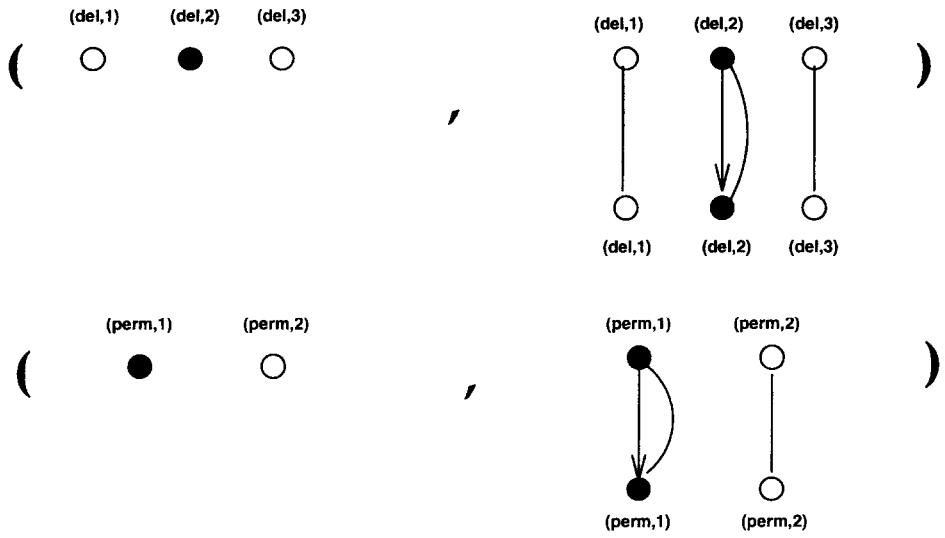

FIGURE 4. Circular variants for the perm example.

them. From the resulting set, circular variants are constructed. They are shown in Figure 4. The condition of Theorem 3.6 is fulfilled so that we may conclude left-termination for the considered queries. □

In conclusion, note that these termination conditions are indeed conceptually similar to those in Section 3.2. However, the transformation to Datalog and the graphical representation of the different concepts and conditions makes a detailed comparison difficult. Observe that a restriction to direct recursion is not needed. Even for mutually recursive Datalog Programs, only a finite number of circular variants exist.

3.5. *Other Approaches*

One of the first contributions to the topic is by Vasak and Potter in [110]. They present a theoretical characterization of the set of universally terminating queries to a program. A key notion is a sequence of sets, $M_n \subseteq Atom_P/\sim$, $n \in \mathbb{N}$, such that M_n contains all atoms for which there exists a proof tree with a depth of at least n. To each of these sets, we can associate its complement, co-$M_n = Atom_P/\sim - M_n$, where $-$ stands for the set difference operator. The terminating queries can then be characterized as $\cup_{n\in\mathbb{N}}$ co-M_n.

The paper presents a formalization of the termination problem in terms of fixpoint characterizations, but it provides little or no support for—even manual—termination verification.

Another theoretical characterization of terminating and nonterminating queries is described in [14]. In this work, a first-order formula is associated to each program: the *completion formula of finiteness*. It is proved that ground valid consequences of this formula cannot have any infinite SLDNF-computation. A fixpoint characterization is provided to identify the ground atomic consequences. Secondly, a *completion formula of infiniteness* is constructed for each program. This formula is expressed in the language of first-order modal logic of provability (see [25]). Ground valid consequences of this formula may have

an infinite SLDNF-computation. Because modal logic of provability is not axiomatizable [25], it is, in general, not possible to conclude which atoms are consequences of this formula.

[65] deals with existential termination. The property is proved by constructing proof rules based on the notion of *parametrized invariants*. Sequences of computation states are mapped into a decreasing sequence of a well-founded set by means of a *variant function*. However, as the execution states and transitions are described by proof rules, the approach is rather complex, even for very simple programs.

Another alternative for proving termination is to use structural induction proofs. As an example, we can prove left-termination of permute for all queries $\leftarrow permute(t_1, t_2)$, with t_1 a list of fixed length, by first proving left-termination for the case that t_1 is the empty list, and then proving that left-termination for a list t_1 of length n follows from the assumption of left-termination for lists t_1' of length smaller than n.

[42] and [18] propose techniques of this type. We briefly comment on the latter. [18] is a transformational approach. It defines a semantics for Prolog, including features as cut and negation as finite failure, by assigning a function $[\![p]\!]$ of arity n, to every predicate p of arity n. The function $[\![p]\!]$ maps each n-tuple of terms, $\bar{t}$, to a sequence of answer substitutions. The sequence is identical to the sequence of computed answers generated for $\leftarrow p(\bar{t})$ under the Prolog inference rule. Then, [18] uses inductive termination proof techniques, for Functional Programs, adapted from the work of Cartwright and McCarthy [37–39], to prove the termination or nontermination of the Prolog program.

The technique is very general. It can deal with indirect recursion, normal programs, both existential and universal termination, and with impure features of Prolog. On the other hand, with respect to automation, it requires the use of general inductive theorem proving techniques. It is unclear to us whether this can lead to more powerful automated termination analysis than with other techniques presented in the literature. We refer to [18] for an explicit treatment of the permute example.

Similarly, [42] uses structural induction to prove a variant of the existential termination property. As complete abstraction is made of the selection rule, termination means the existence of *a solution* in some SLD-tree.

4. SOME SPECIFIC TECHNICAL ISSUES

In this section, we survey contributions to LP-termination analysis that are related to more specific technical issues than those addressed in the previous section. We have grouped them as follows. Section 4.1 contains additional concepts and techniques related to norms, level mapping, and well-founded orderings. In Section 4.2, we survey several approaches for inferring interargument relations. Section 4.3 is on how techniques tuned on recursion deal with mutual recursion.

4.1. Norms, Level Mappings and Well-Founded Orderings

We address three issues related to the study and the selection of useful well-founded orderings. First, we devote considerable attention to the study of various norms. The use of norms as a basis for defining well-founded orderings is rather specific for LP termination analysis. In Term Rewriting, well-founded orderings are most often defined through syntactic subterm orderings. A next issue is to provide norms that are suitable for appropriately ordering partially instantiated terms. Here, dealing with the backpropagation problem is

again the main concern. Finally, we discuss the problem of how to select a well-founded ordering, adequate for proving termination of a program at hand.

The work of Naish in [86] and [87] has had a significant impact on LP termination analysis. The work addresses control generation, with an emphasis on the avoidance of infinite derivations. Although the analysis is at the clause level and mostly of a heuristic nature rather than providing formal termination conditions, the technique allows us to avoid many infinite derivations in practice, and was a source of inspiration and motivation for many other works.

The main drawback of Naish's approach is that his well-founded orderings are based on the syntactic subterm order. Ullman and Van Gelder's work [107] was motivated by the fact that such a subterm order is incapable of adequately dealing with local variables: such variables have no syntactic relation to the head of the clause. As far as we know, they were the first to introduce an abstraction function, mapping terms to their sizes, as a basis for the well-founded ordering in a top-down termination analysis. Their abstraction function is list-length. Later, the concept was generalized to that of a norm.

Definition 4.1. (norm)
A norm is a function $||.|| : Term_P/_\sim \rightarrow I\!N$. □

With slight abuse of notation, we also write $||t||$, for $t \in Term_P$. Examples of norms that are often used in practice are *list-length* and *term-size*. Term-size counts the number of function symbols in a term.

Definition 4.2. (term-size)
The *term-size* norm, denoted $||.||_t$, is defined in the following way:

$$||f(t_1, \ldots, t_n)||_t = 1 + \sum_{i=1}^{n} ||t_i||_t \quad \text{with } f \text{ any function symbol and } n > 0$$
$$||x||_t = 0 \quad \text{otherwise.} \quad \square$$

Another frequently used norm is *term-depth*, which gives the maximum depth of (the tree representation of) a term.

Definition 4.3. (term-depth)
The *term-depth* norm, denoted $||.||_d$, is defined in the following way:

$$||f(t_1, \ldots, t_n)||_d = 1 + max_{1 \leq i \leq n} ||t_i||_d \quad \text{with } f \text{ any function symbol and } n > 0$$
$$||x||_d = 0 \quad \text{otherwise.}$$

In [90], [91], Plümer introduces the class of *linear norms*.

Definition 4.4. (linear norm)
A norm $||.||$ is linear if, for each term $t \in Term_P$ and for each substitution σ such that $t\sigma$ is ground, we have

$$||t\sigma|| = ||t|| + \sum_{i=1}^{n} ||v_i\sigma||$$

where $\{v_1, ..., v_n\}$ is the multiset of variable occurrences in t. □

Term-size is linear; list-length and term-depth are not.
In [26], the more general class of *semi-linear norms* is introduced.

Definition 4.5. (semi-linear norm)
A norm $||.||$ is *semi-linear* if it is recursively defined by means of the following schema:

$$\begin{aligned} ||V|| &= 0 && \text{if } V \text{ is a variable, and} \\ ||f(t_1, ..., t_n)|| &= c + ||t_{i_1}|| + ... + ||t_{i_m}|| && \text{with } c \in \mathbb{N},\ c, i_1, ..., i_m \text{ only} \\ & && \text{dependent on } f/n, \text{ and} \\ & && 1 \leq i_1 < ... < i_m \leq n. \quad \square \end{aligned}$$

Linear norms are semi-linear. Again, term-depth is an exception. The relevance of semi-linear norms relates to the notion of rigidity. In Section 3.2, we introduced this notion for atoms with respect to a generalized level mapping. It was originally introduced in [26] as a property of terms with respect to a norm.

Definition 4.6. (rigid term)
Let $||.||$ be a norm and $t \in Term_P$. We say that t is rigid with respect to $||.||$, if for any substitution θ, $||t\theta|| = ||t||$. $\square$

Since level mappings are usually inferred from norms by computing a sum or positive linear combination of the norms of the terms occurring on some fixed argument positions of the atoms, the two rigidity notions on atoms and terms are closely linked. An atom is rigid with respect to such a level mapping if all terms occurring on the selected argument positions of the atom are rigid with respect to the underlying norm.

In general, it is difficult to decide which terms are rigid. This is precisely why semi-linear norms were introduced: terms that are rigid with respect to a semi-linear norm can be syntactically characterized. Essentially, a term is rigid with respect to a semi-linear norm $||.||$ if and only if recursively decomposing the term over the relevant argument positions $i_1, ..., i_m$ for its principle functor (see Definition 5) does not produce any variables. In other words, no variable can occur at any (nested) relevant argument position of the given norm (see [26] for a more formal treatment).

Example 4.1.
Let $f/3$ and $g/2$ be the functors in the language underlying to P. Let $||.||$ be the semi-linear norm

$||x|| = 0$ if x is a variable
$||f(t_1, t_2, t_3)|| = 1 + ||t_1|| + ||t_3||$
$||g(t_1, t_2)|| = ||t_1||$
$||a|| = 0$
$||b|| = 0$

Then, $g(f(a, g(x, y), g(b, x)), z)$ is rigid. The norm of any of its instances is 1. The term $g(f(x, a, b), c)$ is not rigid. Taking $x = f(a, a, a)$, its norm is 2; taking $x = a$, its norm is 1. $\square$

Most of the techniques that do not impose the restriction of well-modedness study termination for queries that are characterized through rigidity of certain arguments with respect

to a semi-linear norm. As such, these notions play a central role in several recent works (e.g., [93, 113]).

Using a semi-linear norm and given a functor f, the norm relates the size of all terms with top-level functor f in the same way to the size of its subterms. This is not always convenient. As a hypothetical example, if, in the same program, both terms of the form $f(t_1, f(t_2, f(...)))$ and terms of the form $f(f(f(..., t_2), t_1)$ are processed, then it may be desirable to measure them in different ways. Terms of the first type could be measured by $||f(t_1, t_2)|| = 1 + ||t_2||$, while terms of the second type are measured by $||f(t_1, t_2)|| = 1 + ||t_1||$. In [26], semi-linear norms are generalized to *typed norms*, allowing this increased flexibility. Also, [54] introduces more general norms with a similar expressivity. In addition, the latter work presents a technique for automatic generation of suitable norms, using type inference.

Even when a norm has been fixed, one is still faced with the problem of defining a suitable generalized level mapping on the basis of the norm. In the context of well-moded programs, a straightforward solution is to define for each predicate symbol p of arity n in P:

$$f(p(t_1, ..., t_n)) = \sum_{i \in I_n} ||t_i||$$

where I_n is the set of all input arguments according to the mode of p. However, this approach is insufficiently refined, as the following example illustrates.

Example 4.2. (reverse)

reverse(Nil, x, x) ←
reverse($[x|y], z, u$) ← reverse($y, z, [x|u]$).

Assume that we aim to prove termination for the queries specified by the mode reverse(in, out, in). The program is well-moded with respect to this mode. With list-length as the selected norm and following the above construction for the level mapping, we get f(reverse(t_1, t_2, t_3)) = list-length(t_1)+list-length(t_3) Proving termination on the basis of this level mapping is impossible since, for any derivation, it assigns the same value to every reverse atom in it. □

The problem is, of course, that the third argument of reverse should not be taken into account. To solve the problem, one could consider all possible subsets of input positions for each predicate, and attempt to prove termination for each resulting level mapping. This, of course, strongly increases the complexity of the analysis. Sohn and Van Gelder in [105] propose a solution based on linear programming techniques. In the context of the example, they suggest a symbolic level mapping of the form f(reverse(t_1, t_2, t_3)) = a·list-length(t_1) + b·list-length(t_3), where a and b are variables ranging over the natural numbers. The problem is to assign values to a and b such that the resulting level mapping can be used to prove termination.

Computing such values is done using linear programming. A termination condition similar to the ones expressed in Section 3.2 is formulated as a system of linear inequalities over the variables in the symbolic level mapping. For each recursive clause, an inequality expresses that, for each instance of the clause, the symbolic level mapping of the head

should be larger than that of a recursive call. In our example, we get the inequality

$$a(1 + y') + bu' > ay' + b(1 + u')$$

Here, the variables y' and u' are abstractions for the values that the list-length norm would produce for any specific instance of the variables y and u. The inequality has a minimal solution: $a = 1$ and $b = 0$. Obviously, the resulting level mapping decreases at every derivation step, thereby proving termination.

Our presentation of [105] only sketches the main intuitions. Many aspects are omitted. Dealing with left-termination and local variables is solved by adding interargument relations to the system of inequalities.

We conclude the subsection with some comments on a very different approach, both for providing well-founded orderings and for dealing with the backpropagation problem. It was recently proposed in [33].

Instead of using norms and level mappings, this technique relies on general term orderings, as is done in Term Rewrite Systems. It imposes no restriction of groundness, boundedness, or rigidity of the input. To deal with backpropagation, [33] assumes that refined type information on the calls is available, and that this type of information satisfies a generalized notion of well-modedness, not restricted to ground input and output. Then, the program is transformed. The head of every clause is compared with the call type for the corresponding predicate. If the head contains terms that are more specific than the call type, then these more specific parts are generalized. Finally, the termination conditions impose a decrease under the term ordering for the transformed clauses.

The result of the transformation is that only part of the structure occurring in the head of the original clauses is taken into account to prove well-foundedness, namely, the part that is guaranteed to consume data from the call. Parts that may construct data in some calls are disregarded.

This is very similar to the approach of Naish in [87]. The purpose of his wait-declarations is precisely to delay unifications that cause construction of data in the call. Calls that are less specific than the head are delayed.

Whether the approach of [33] can adequately deal with local variables, without additionally incorporating interargument relations computed through norms, is unclear at this stage. We return to this point in the next subsection.

4.2. *Deriving Interargument Relations*

In this subsection, we review several techniques for deriving interargument relations.

Definition 4.7. (interargument relation)

Let P be a definite program, p/n a predicate in P, and $||.||$: $\text{Term}_P/_\sim \rightarrow \mathbb{N}$ a norm. An *interargument relation for p wrt* $||.||$ is relation $R_p \subseteq \mathbb{N}^n$, such that for every computed answer, $p(t_1, ..., t_n)$, for any possible call to p, $(||t_1||, ..., ||t_n||) \in R_p$. □

Example 4.3. (delete)

For the delete predicate and the list-length norm, some interargument relations are $\{ (x, y, z) \in \mathbb{N}^3 \mid y = z + 1 \}$, $\{ (x, y, z) \in \mathbb{N}^3 \mid y > z \}$, and $\{ (x, y, z) \in \mathbb{N}^3 \mid y > 0 \}$. The first of these three is more precise than the others, in the sense that the others are only supersets of the set of n-tuples $(||t_1||, ..., ||t_n||)$ that can occur for computed answers $p(t_1, ..., t_n)$.

Some works introduce a notion of interargument relation which is less restrictive than the one above. In particular, the condition $(||t_1||, ..., ||t_n||) \in R_p$ is often only required for atoms $p(t_1, ..., t_n)$ in the Least Herbrand model of P.

Notice that the latter definition generalizes the previous one due to a completeness result for SLD-resolution: any ground atomic consequence of P has an SLD-refutation (see, e.g., [78]). The latter definition can also be alternatively formulated as follows: a relation $R_p \in \mathbb{N}^n$ is an interargument relation for a predicate p/n of P if there exists a model, I, of P such that the domain of I is $\mathbb{N}$, the preinterpretation of I is (in a natural way) induced by the restriction of $||.||$ to the ground terms in Term_P, and $R_p = I(p)$. We omit the proof of the equivalence of these formulations. Our main reason for stating the equivalence is to establish the connections with the termination conditions expressed in Section 3. Here, in the notions of acceptability and acceptability with respect to a set of atoms, interargument relations are further generalized to *any models* of the program. In this last generalization, the restrictions on the domain of the interpretation and on the preinterpretation have disappeared.

Several formulations in the literature are often more restrictive than Definition 7 on the level of the type of relation R_p that is allowed. For instance, the technique of [107] only allows relations of the form $||t_i|| + c \geq ||t_j||$, for some integer constant c and two argument positions i and j. This technique is particularly restrictive in this respect. At the other end of the spectrum, [109] allows relations R_p/n that are the integer solutions of systems of linear inequations over n variables. These differences in expressivity are surveyed in detail in the following subsections, in which our main focus is on presenting the main techniques proposed for inferring the relations.

As mentioned in Section 1.4.3, the motivation for deriving such relations is the treatment of local variables in the context of left-termination. Returning to the permute example, in order to prove a decrease of the list-length norm between the first argument of the head, $[x_1|y_1]$, and that of the permute atom in the body, w_1, of

$$\text{permute}([x_1|y_1], [u_1|v_1]) \leftarrow \text{delete}(u_1, [x_1|y_1], w_1), \text{permute}(w_1, v_1)$$

we restrict our attention to instances of the clause in which $\text{delete}(u_1, [x_1|y_1], w_1)$ is a computed answer for some query to delete. Then, both the first and the second interargument relations for delete in Example 4.3 are sufficient to prove the decrease.

One might observe that, in view of this motivation, it is unclear why increased expressivity, such as solutions to systems of linear inequations, is useful. The final purpose of the interargument relation is to prove a simple inequality between the sizes under some norm of some argument positions anyway.

The reason is that, although in the termination condition we might only need a simple inequality, relating two argument positions, there may be many intermediate predicates involved in linking the head atom's arguments to the local variable in the recursive call. The interargument relations for all these predicates need to be computed and combined to obtain the desired inequality in the termination condition. By restricting the expressivity with which the interargument relation of each separate predicate involved in this link is represented, their combination may become too imprecise to conclude a decrease.

We illustrate the point with a simple (although not very practical) example.

Example 4.4. (confused delete)

$$\begin{aligned} \text{conf}(x) \quad &\leftarrow \text{delete}_2(x, z), \\ &\quad\ \text{delete}(u, y, z), \\ &\quad\ \text{conf}(y). \\ \text{delete}_2(x, y) \ &\leftarrow \text{delete}(u, x, z), \\ &\quad\ \text{delete}(v, z, y). \end{aligned}$$

Procedurally, using the left-to-right selection rule, during every recursion cycle, two members of a list, x, are removed to obtain z and a new member, u, is added to z to obtain y, which is then processed in the next recursive cycle. Under the list-length norm, a precise interargument relation for delete_2 is $\{(x, y) \in \mathbb{N}^2 \mid x = y+2\}$. For delete, a precise relation is $\{(x, y, z) \in \mathbb{N}^3 \mid y = z + 1\}$. Combining these relations allows us to conclude that in any instance of the recursive clause for conf/1, such that the delete_2 and the delete atom are computed answers, list-length(y) < list-length(x). Thus, left-termination is proved.

If we were only offered the expressivity of interargument relations of the form $||t_i|| > ||t_j||$, then the most precise interargument relation for delete_2 and delete would, respectively, be $\{(x, y) \in \mathbb{N}^2 \mid x > y\}$ and $\{(x, y, z) \in \mathbb{N}^3 \mid y > z\}$. As a result, no conclusion on the relation between the size of the instances of y and x could be drawn. □

Although the example is not convincingly practical, the point is that reduced refinement in the interargument relations can—due to propagation of the relations throughout the derivation—often degenerate below any point of usefulness. This is a main reason why norms have been used in this context. By ordering atoms via natural numbers assigned to their terms, the gain is the arithmetic structure over the natural numbers. In the context of syntactic subterm orderings, this advantage cannot be obtained. This motivates our comment at the end of the previous Section in relation to [33].

The remainder of the subsection is organized as follows. We start with two technically very similar approaches, one by Ullman and Van Gelder [107], the other by Plümer [90]. Then we sketch an approach based on abstract interpretation by Verschaetse and De Schreye [53] and, again, a related one by Brodsky and Sagiv [32]. We end with some comments on alternative approaches by Van Gelder [109], Cousot and Cousot [46], and Mesnard and Ganascia, [85].

4.2.1. An Approach Based on Variable/Argument Graphs. Ullman and Van Gelder were the first to address and successfully tackle the problem. Their approach [107] is of somewhat restricted applicability, in the sense that it only allows us to derive interargument relations that take the form of inequalities between the sizes of two argument positions. More precisely, they compute relations of the form

$$p_i + c \geq p_j$$

where p_i and p_j are so-called *argument designators* and c is an integer. An argument designator, p_i, is a variable denoting the size-abstraction of any term that can occur at the ith argument position of a computed answer $p(t_1, ..., t_n)$. In [107], "size" refers to the terms value under the list-length norm.

Efficiency has been a main concern in the design of the method. The analysis is meant to be effected *at run-time* within the NAIL! system, providing the system with information to control its derivations. As a result, a polynomial time algorithm is required. Both the

restricted expressivity of the derived relations and the restriction to list-length are limitations imposed in order to achieve this desired efficiency. The purpose of the latter restriction is that, within a clause, the size of any argument can be related to at most one variable appearing in the clause: namely, for an open-ended list, the tail of the list. This avoids a search over various possible relations.

The analysis is top-down, in the sense that the derivation of an interargument relation for a predicate p will activate the analysis for all predicates on which p depends. The basic concept is the *variable/argument (V/A) graph*. Given a clause, its V/A graph represents relations between sizes of arguments and sizes of variables occurring in the clause. The graph contains a node for each argument position of each atom in the clause. It also contains a node for each variable. A labeled arc connects an argument node to a variable node if the argument contains an open-ended list and the variable is the tail of the list. The arc is labeled by their difference in size. The relation expressed in the arcs is transitive. Exploiting the transitivity allows us to infer relations between the size of different arguments of an atom: the desired interargument relation.

More formally, a V/A graph is defined as follows.

Definition 4.8. (basic V/A Graph)

For each clause $A \leftarrow B_1, ..., B_m$ in a program P, there is a corresponding (*basic*) *V/A graph*. Nodes in this graph are defined as follows.

- For each argument of the clause head A, there is a node with label p_i' $(1 \leq i \leq n)$, where p/n is the predicate symbol of A. The label p_i' is primed in order to distinguish it from a corresponding p_i in the body.
- For each variable appearing in the clause, there is a node labeled by that variable.
- For each argument of an atom B_j in the body of the clause (with corresponding predicate symbol q^j/n_j), there is a node whose label is an unprimed argument designator $q_i^j (1 \leq i \leq n_j)$. In case there is more than one atom with the same predicate symbol in the body, corresponding labels are distinguished by adding extra superscripts.
- There is one special node labeled 0, which is needed to represent the size of constant terms.

For each node corresponding to an argument position p_i, there is an arc to the node V if p_i corresponds to a list with tail V. It has the label $-d$ if d is the difference in list-length between p_i and V. There is also a corresponding arc from V to p_i labeled $+d$. If p_i stands for a nil-terminated list, there is instead an arc to the 0-node, labeled with $-d$, if d is the list-length of p_i and vice versa.

All variables have an unlabeled arc going to the 0-node, formalizing the idea that their size is at least zero. □

Example 4.5. (permute)

In Figure 5, the basic V/A graph for the second clause of the permute predicate is shown. For simplicity, the 0-node and its connecting arcs are omitted.

We explain the way in which the interargument relation is inferred from the V/A graph on the basis of our example. Assume that we aim to prove a relation of the form $perm_1 + x_1 \geq perm_2$. This can be achieved by finding a path in the graph connecting node $perm_1'$ to node $perm_2'$ and by adding up the labels on this path. By the assumption that the relation

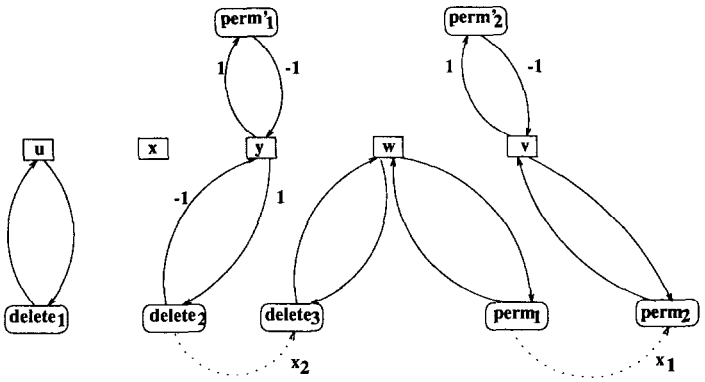

FIGURE 5. Basic V/A graph for the recursive perm clause.

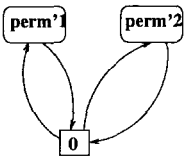

FIGURE 6. V/A graph for the clause perm(Nil,Nil).

$perm_1 + x_1 \geq perm_2$ holds for all correct answers permute(u_1, u_2), we may connect node $perm_1$ to node $perm_2$ by an arc labeled by x_1.

Then we try to add new arcs to the graph, such that a path from $perm'_1$ to $perm'_2$ is obtained. A restriction on adding such new arcs is that they should correspond to potential interargument relations for the predicates on which permute depends. In the example, the only possibility that produces a path is to add an arc connecting node $delete_2$ to node $delete_3$. We label it with a variable x_2. Adding this labeled arc corresponds to a new hypothesis that there exists an interargument relation for the delete predicate of the form $delete_2 + x_2 \geq delete_3$.

Under the assumption that this hypothesis holds for some x_2, adding up the labels for the path connecting $perm'_1$ to $perm'_2$, we get the new inequality: $perm_1 + x_2 + x_1 + 1 \geq perm_2$. If we want the inequality $perm_1 + x_1 \geq perm_2$ to be a safe approximation of the relations expressed in all clauses, we can derive: $x_1 \geq x_2 + x_1 + 1$.

Then, the same reasoning is applied to the V/A graph for the nonrecursive clause for permute. Its V/A graph is shown in Figure 6. We derive a relation $perm_1 \geq perm_2$. Again, since $perm_1 + x_1 \geq perm_2$ must safely approximate all relations, the constraint $x_1 \geq 0$ immediately follows.

The next step is to start the same analysis for the delete predicate and the assumption $delete_2 + x_2 \geq delete_3$. Both V/A graphs for the clauses for delete contain a path connecting $delete'_2$ to $delete'_3$. Only the clause for the base case imposes a constraint $x_2 \geq -1$.

Finally, all constraints are combined in a system of inequalities:

$$\begin{cases} 0 \geq x_2 + 1 \\ x_1 \geq 0 \\ x_2 \geq -1 \end{cases}$$

The aim is to obtain a minimal solution for x_1 since any $x_1' > x_1$ will trivially also satisfy $perm_n + x_1' \geq perm_2'$.

Ullman and Van Gelder define a fixpoint operator to compute the minimal solution. The operator requires at most n iterations if n is the number of symbolic constants (e.g., x_1 and x_2) in the constraint set. The resulting interargument relations are proven to hold for all atoms in the least Herbrand model. In the example, the minimal solution is reached for $x_1 = 0$ and $x_2 = -1$. The corresponding relations are $perm_1 \geq perm_2$ and $delete_2 - 1 \geq delete_3$. □

As mentioned, the main concern in [107] is efficiency of the analysis. To this purpose, they impose an additional *uniqueness restriction* on the program clauses. Uniqueness of a clause forbids the existence of more than one path between any two nodes corresponding to arguments in the head. This transforms the combinatorial problem of guessing what set of inequalities will be needed into a deterministic one. On the level of the program, uniqueness essentially means that each variable has at most one producer (see Section 2).

4.2.2. AN ANALYSIS BASED ON AND/OR DATAFLOW GRAPHS. The two most important restrictions imposed in the previous techniques are removed in the approach of Plümer [90]. On the level of expressivity, observe that even for the very simple case of append, the V/A graph approach only allows us to derive the interargument relations $append_3 \geq append_1$ or $append_3 \geq append_2$. A more expressive relation would be $append_3 \geq append_2 + append_1$. In [90–92], Plümer extends the work of Ullman and Van Gelder by allowing more general *linear predicate inequalities* of the form

$$\sum_{i \in I} p_i + c \geq \sum_{j \in J} p_j,$$

where I and J are, respectively, subsets of the sets of input and output positions for the predicate, p_i and p_j are argument designators, and the offset $c \in \mathbb{Z} \cup \{\infty\}$. Input and output positions are specified using mode information. In [90] and [91], all input is assumed ground, and a successful computation binds all output variables to ground terms. In more recent work [92], the technique is extended by allowing rigid terms instead of ground terms.

Another difference with respect to Ullman and Van Gelder's technique is the use of more general linear norms (see Section 4.1). Similar to the motivation for the selection of list-length in [107], linear norms allow us to relate the size of a term to the size of any variable occurring in it. Recall that list-length is not linear. In [92], Plümer extends his technique by incorporating semi-linear norms.

Essential in Plümer's approach is considering *sets* of argument positions instead of single argument positions. The entire generalization over [107] is very similar to the way in which the problem of finding a *solution graph* in an And/Or graph generalizes the path-finding problem for graphs (see, e.g., [89]). This motivates the notion of an *And/Or dataflow graph*.

Definition 4.9. (And/Or dataflow graph)

Let $G = A_1, \ldots, A_n$ be a conjunction of atoms. Let $In = \{u_1, \ldots, u_m\} \subseteq Var_G$ denote the set of variables that occur on input positions and $Out = \{w_1, \ldots, w_k\} \subseteq Var_G$ the set of variables occurring on output positions. For every atom A_i, there is a corresponding linear predicate inequality LI_i $(1 \leq i \leq n)$. An *And/Or dataflow graph* for G is an And/Or graph, which is constructed as follows.

Nodes:

- For each variable $v \in Var_G$, there is a node labeled v. Such nodes are called *Or-nodes*.
- There are two special And-nodes: the "start node" is labelled *Out*, while the "end node" has label *In*.
- For each atom A_i in G, there is a corresponding *And-node*.

Connectors:

- There is a k-connector $(Out, w_1, \ldots, w_k)$.
- For $1 \leq i \leq m$, there are 1-connectors (u_i, In).
- Let n be the node representing atom A_i in G, LI_i is the corresponding linear predicate inequality, and $V = \{v_1, \ldots, v_r\}$ and $V' = \{v'_1, \ldots, v'_s\}$ are the sets of variables occurring on, respectively, the input and the output positions of A_i with respect to LI_i. There is an r-connector $(n, v_1, \ldots, v_r)$, and for all $v' \in V'$, there is a 1-connector (v', n). If $V = \emptyset$, then an additional dummy node 1_V is added, together with the 1-connector $(1_V, In)$. □

Example 4.6. (permute)
The And/Or dataflow graph for the body of the recursive clause for permute is shown in Figure 7. Specification of the mode permute(in,out) gives the sets $In = \{x, y\}$ and $Out = \{u, v\}$. □

Starting from an And/Or dataflow graph, several solution graphs (subgraphs) can be obtained by choosing nondeterministically one descendant in each Or-node.

In Figure 7, the And/Or dataflow graph forms its own (unique) solution graph since every Or-node has exactly one descendant.

For purely technical reasons, Plümer imposes some restrictions. The important one is that his algorithms require solution graphs to be *admissible*, i.e., each Or-node must have exactly one ingoing connector. This comes down to restricting the technique to clauses that have only one consumer for each variable that acts as a producer (see Section 2).

Another important concept is the *weight of an admissible solution graph*. It generalizes the result of summing up the labels on the path connecting p'_i to p'_j found in the previous approach. It is computed as the sum of the weights of each And-node in the graph, where the weight of an And-node corresponding to an atom $A_k = p(t_1, \ldots, t_n)$ is on its turn defined as

$$\omega_k = \sum_{i \in I} ||t_i|| - \sum_{j \in J} ||t_j|| + c,$$

where $LI_k = \sum_{i \in I} p_i + c \geq \sum_{j \in J} p_j$ is the corresponding linear predicate inequality for A_k.

The admissible solution graphs are both used for verifying termination conditions for the recursive clauses and for deriving valid linear predicate inequalities. Mutual recursion is excluded. This enables the processing of all predicates in a purely bottom-up fashion,

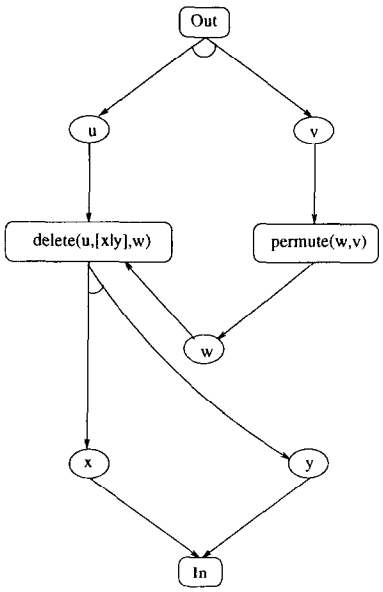

FIGURE 7. And/Or dataflow graph for the recursive clause of permute/2.

using the termination conditions and predicate inequalities that are obtained for lower level predicates, to derive the same information for the higher situated predicates.

Computation of the weight of the admissible solution graph requires that the linear predicate inequalities for the atoms $A_1, \ldots, A_j$ are known. Plümer gives an algorithm that first constructs all admissible solution graphs for these atoms. The solution graphs themselves give on their turn rise to sets of constraints, which are solved using a specialized algorithm. The optimal solutions finally yield the desired linear predicate inequalities.

As formulated in [90] and [91], the technique is deliberatively restricted to directly recursive programs, relying on program transformation (see Section 4.3) to deal with the mutually recursive case. In [67], it is extended to deal with mutual recursion directly. The main problem involved in the extension is that the derivation of an interargument relation for a mutually recursive predicate, on the basis of the interargument constraints local to each clause, is in general NP-complete. It is shown that, due to the special form of the constraints, the algorithm presented in [67] is polynomial.

4.2.3. DERIVING LINEAR SIZE RELATIONS BY ABSTRACT INTERPRETATION. The expressivity of Plümer's interargument relations has been further improved upon. Consider the following program.

Example 4.7. (duplicate)

duplicate(Nil, Nil) $\leftarrow$
duplicate($[x|y], [x, x|z]$) $\leftarrow$ duplicate(y, z)

The list in its second argument contains a duplication of every member of the list in its first argument. Using list-length, the most precise interargument relation that can be obtained is $\text{duplicate}_2 = 2{\cdot}\text{duplicate}_1$. Neither of the previous approaches supports this

type of expressivity. In the case of Plümer's approach, the limitation of not being able to express linear combinations of argument designators is directly related to the admissibility restriction. □

[112] presents a technique based on abstract interpretation to infer interargument relations of the form $c_0 + \sum_{i=1}^{n} c_i p_i = 0$, where $c_i, i = 0, ..., n$, are integers. The technique is extended in [113], where *conjunctions* of such linear equations are derived. They are referred to as *linear size relations*. The advantage of allowing conjunctions is illustrated by the following program, which defines the addition of two vectors.

Example 4.8. (sum)

$\text{sum}(Nil, Nil, Nil) \leftarrow$
$\text{sum}([x_1|y_1], [x_2|y_2], [x_3|y_3]) \leftarrow x_3 \text{ is } x_1 + x_2, \text{sum}(y_1, y_2, y_3)$

The most precise interargument relation here is $\text{sum}_1 = \text{sum}_3 \wedge \text{sum}_2 = \text{sum}_3$. □

The main contribution of [113] is that the technique is not a special purpose analysis designed for inferring interargument relations, as is the case for the two previous approaches. The technique is formulated as an instance of a generic framework for abstract interpretation. The framework is the one of Bruynooghe [35], which is a top-down analysis based on abstract AND–OR-trees.

The main advantage of inferring the interargument relations by means of a generic analysis tool is that it simplifies the correctness proof. The abstract interpretation framework itself guarantees correctness of any application developed within it, provided that the specific abstract domain satisfies some properties and that certain basic operations over this domain are specified and proved correct.

Not surprisingly, the abstract domain for the application consists of systems of linear equations over the variables of the clauses. Given that the number of variables in a clause under consideration is n, such systems can be geometrically interpreted as affine subspaces of $\mathbb{R}^n$, intersected with $\mathbb{N}^n$. Every individual linear equation corresponds to a hyperplane in this space. This interpretation is exploited for defining the structure of the domain (a partial order with a minimal and maximal element, an upper bound operation, and a condition on the absence of infinite ascending chains under this order) and the basic operations required in the framework (initialization, procedure entry, procedure exit, and the interpretation of built ins). Each of these can be related to properties or operations known from linear algebra. With respect to the operations, they are defined in terms of the intersection, union, projection, and extension operators for affine spaces, defined by Karr in [72]. Again, this facilitates the correctness proofs.

As a last comment on the technique, no restrictions such as uniqueness or admissibility are imposed. The only requirement is that the used norm is semi-linear.

4.2.4. BOTTOM-UP INFERENCE OF DISJUNCTIVE CONSTRAINTS. For some programs, all previous approaches still lack expressivity. Consider merge.

Example 4.9. (merge)

$$\begin{aligned}
&\text{merge}(Nil, x, x) &&\leftarrow\\
&\text{merge}(x, Nil, x) &&\leftarrow\\
&\text{merge}([x|x_s], [y|y_s], [x|z_s]) &&\leftarrow x \leq y, \text{merge}(x_s, [y|y_s], z_s)\\
&\text{merge}([x|x_s], [y|y_s], [y|z_s]) &&\leftarrow y < x, \text{merge}([x|x_s], y_s, z_s)
\end{aligned}$$

In a termination proof, we may require an interargument relation for merge, expressing that, for nonempty lists, merge_3 is strictly larger than either merge_2 or merge_1:

$(0 = \text{merge}_1 = \text{merge}_2 = \text{merge}_3) \vee (\text{merge}_3 \geq \text{merge}_2 + 1) \vee (\text{merge}_3 \geq \text{merge}_1 + 1)$

The information cannot be expressed using the previous approaches. □

In [32], Brodsky and Sagiv present a technique for deriving disjunctions of conjunctions of inequalities of the form $p_i + k \geq p_j$. They are referred to as *disjunctive constraints*. The method is bottom-up and expressed in terms of Datalog programs.

It starts from a set of given inequality constraints for all EDB relations, and iteratively derives disjunctive constraints for the IDB relations.

The basic tool for deriving inequalities is a graph which resembles the V/A graph of Ullman and Van Gelder: the *characteristic graph*, which is constructed local to each clause. Its main purpose is to relate the sizes of the different variables. Therefore, such a graph contains nodes for all variables in the clause and includes relationships between them, induced by the constraints which are known for the body atoms. If $S(\bar{x})$ is an atom of the body and $p_i + k \geq p_j$ is a valid inequality constraint for the S predicate, then the characteristic graph contains an arc with weight k from variable node x_i to node x_j. To find an inequality between two variables x_i and x_j in argument positions i and j of the head, the minimum path between them must be determined. Supposing it has weight w, the authors prove that the inequality $p_i + w \geq p_j$ is a valid inequality for the predicate.

Note that for predicates in the body, disjunctive constraints may be available. Each of these disjuncts should be considered separately, thereby leading to disjunctive constraints for the clause head predicate.

To generate all disjunctive constraints, Brodsky and Sagiv introduce an immediate inference operator I_P. Initially, all disjunctive constraints for all IDB predicates are defined to be false (empty disjunction) and disjunctive constraints for all EDB predicates are assumed to be available. At each iteration, each program clause is visited. One disjunct is selected for each atom in the body. All possible relations that can be found between any two argument positions for the head of the clause and for this constraint selection form a new disjunct for this predicate. Valid interargument relations for all predicates are obtained in the fixpoint $I_P \uparrow \omega$.

Example 4.10. (permute)

We illustrate the above with the perm program of Example 3.9.

The analysis requires that for all EDB predicates, inequality constraints are expressed *a priori*. For $P_{[.]}$, we assume that the conjunction $\{P_{[.]1} - 1 \geq P_{[.]3}, P_{[.]3} + 1 \geq P_{[.]1}\}$ is given. P_{nil} trivially has no constraints. Figure 8 shows the characteristic graphs for the del clauses, while those for both perm clauses are pictured in Figure 9. Full lines represent

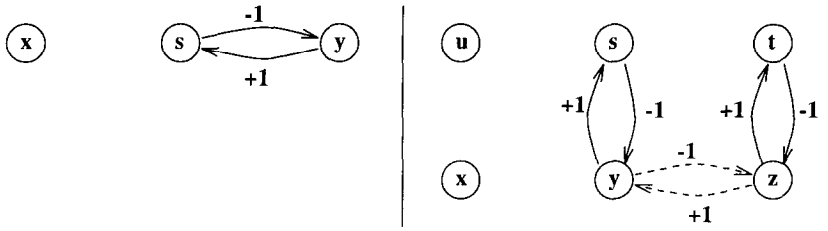

FIGURE 8. Characteristic graphs for the base and recursive clauses for del.

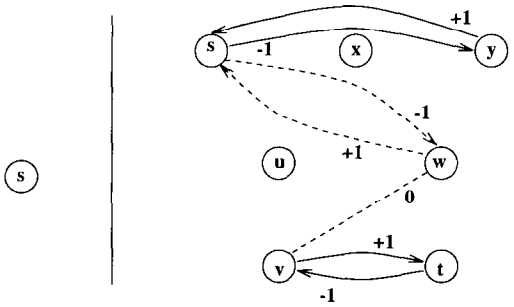

FIGURE 9. Characteristic graphs for the permute clauses.

the information captured in the constraints for the EDB predicates, dashed lines denote relationships that are added during application of the I_P operator.

The first iteration allows us to induce two constraints for the delete predicate: for the base case, in Figure 8, there exists one path from the s-node (corresponding to the second argument) to the y-node (third argument) and vice versa. As yet, the characteristic graph for the second, recursive clause exhibits no paths. In a similar fashion, the characteristic graph for the base case for perm shows two (trivial) paths. Summarizing, $I_P \uparrow 1$ consists of the following constraints: for the delete predicate, the conjunction $\{delete_2 - 1 \geq delete_3, delete_3 + 1 \geq delete_2\}$ and for the perm predicate: $\{perm_1 + 0 \geq perm_2, perm_2 + 0 \geq perm_1\}$. The constraints for the EDB predicates also belong to $I_P \uparrow 1$, but are not shown as they are part of every iteration.

For the next step, I_P is applied to $I_P \uparrow 1$. In Figures 8 and 9, the characteristic graphs corresponding to $I_P \uparrow 1$ now include the dashed lines. Although some new paths appear, no new information is derived, and the operator has reached a fixpoint. □

In general, ω iterations of the I_P operator are needed, thereby causing termination problems for the generation procedure. Also, in the limit, an infinite number of disjuncts could be generated.

To remedy these problems, Brodsky and Sagiv propose to keep track of only one conjunction of constraints instead of multiple ones in disjunction. They introduce a *convex union* operator, which can be seen as an upper bound operation on the domain of their constraint sets. A modified inference operator, S_P, is introduced by constraining the result of I_P to be one conjunct per predicate, taking the convex union with all previously derived

sets. By further imposing a *uniqueness condition*, which is weaker than the one of Ullman and Van Gelder, the authors can guarantee termination of their generation scheme. The resulting algorithm is proven to be polynomial.

To guarantee termination when generating disjunctive constraints, it is shown that a hard condition of *strong uniqueness* must be imposed on the program. A combined approach is also proposed, where n disjuncts are considered, with $n \geq 2$. Two of these disjuncts act as, respectively, a lower bound and an upper bound disjunct.

It is interesting to note that the different algorithms proposed here could easily be mimicked as instances of the bottom-up abstract interpretation framework of [17]. As such, the technique can be regarded as similar to the previous one.

To conclude, we comment on some alternative approaches. In [109], Van Gelder presents an approach which improves on the expressivity of [113] by allowing systems (conjunctions) of *linear inequalities* in addition to systems of linear equations. Note that for any program in which an arbitrary number of members of input lists are disregarded in producing the members of the output lists, the interargument relation can typically not be represented using only equations.

In terms of a geometrical interpretation, systems of linear inequalities correspond to convex polyhedrons. This increases the mathematical machinery needed to process such relations. One basic operation which is essential is the computation of a convex hull.

As in [32], the method is bottom-up and characterizes the interargument relation as a fixpoint of an immediate inference operator. No general method for deriving the fixpoint is provided. Instead, the paper presents a practical heuristic for proposing a fixpoint and a technique for verifying the conjectured solution. In [46], P. Cousot and R. Cousot give a sketch of how abstract interpretation can be used to fully automate the derivation. Their proposal is rooted in earlier work by Cousot and Halbwachs in [47] on the derivation of the same type of interargument relations for imperative programs.

Finally, observe that all the approaches use numerical constraints that are derived locally for each clause and provide some (either top-down or bottom-up) inference scheme to combine, simplify, and propagate these constraints in order to obtain some constraint which holds globally for some predicate. In [85], Mesnard and Ganascia take advantage of the fact that a CLP-language, such as CLP(Q), already provides such an inference engine. They present a technique that transforms a program to a CLP(Q) program in which all the local constraints are made explicit. Then, the interargument relation is returned as a computed answer by executing the CLP(Q) program.

4.3. *Dealing with Mutual Recursion*

The treatment of mutual recursion, in the context of techniques tuned on recursion, is only interesting in view of automation. From a purely theoretical point of view, the problem is trivial. There are at least two obvious solutions to it. A first one is to adapt the termination conditions in Section 3.2 by imposing a decrease of the level mapping, not only for the body atoms with the same predicate symbol as the head, but also for those atoms with a predicate that is mutually recursive with respect to the predicate in the head. By adapting Definition 11 and its refinement to acceptability with respect to a set of atoms in this way, we obtain necessary and sufficient conditions for, respectively, termination and left-termination of any definite program with respect to the set.

Example 4.11. (parsing boolean expressions)

The following program defines the structure of some well-formed Boolean expressions.

$(cl1)$ dis($b_1 \vee b_2$) $\leftarrow$ con(b_1),dis(b_2).
$(cl2)$ dis(b) $\leftarrow$ con(b).
$(cl3)$ con(b_1&b_2) $\leftarrow$ dis(b_1),con(b_2).
$(cl4)$ con(b) $\leftarrow$ bool(b).
$(cl5)$ bool(0) $\leftarrow$
$(cl6)$ bool(1) $\leftarrow$

The program is clearly terminating for the set of all ground queries. The following level mapping can be used in an altered version of Definition 11, along the lines described above, to prove it.

$f(\text{dis}(x)) = 2\times\text{term-size}(x) + 1$
$f(\text{con}(x)) = 2\times\text{term-size}(x)$

All atoms in the bodies of clauses $(cl1)$ – $(cl3)$ are mutually recursive with respect to the head of their clause. The level mapping decreases between each possible call and each of these body atoms in the corresponding instances of clauses $(cl1)$ – $(cl3)$. Thus, the program terminates. □

A second solution is program transformation. There exist several simple techniques to transform mutually recursive programs into directly recursive ones, having identical termination behaviors. The most trivial transformation is to introduce a new predicate symbol, say meta/1, and to replace every atom, A, in the mutually recursive (definite) program by meta(A). The resulting program is directly recursive, so that the termination conditions of Section 3.2 are applicable. More precisely, Propositions 3.2 and 3.3 provide necessary and sufficient conditions for (left)-termination of the transformed program with respect to a set of goals. Furthermore, the two programs have identical termination behaviors: there is a one-to-one mapping relating their sets of SLD-trees, under which an SLD-tree for the original program is transformed by replacing each atom, A, in it by meta(A).

Example 4.12. (parsing Boolean expressions)
Introducing the meta/1 predicate in Example 4.11 gives

meta(dis($b_1 \vee b_2$)) $\leftarrow$ meta(con(b_1)),meta(dis(b_2)).
meta(dis(b)) $\leftarrow$ meta(con(b)).
meta(con(b_1&b_2)) $\leftarrow$ meta(dis(b_1)),meta(con(b_2)).
meta(con(b)) $\leftarrow$ meta(bool(b)).
meta(bool(0)) $\leftarrow$
meta(bool(1)) $\leftarrow$

To show recurrency, the level mapping of Example 4.11 can be reused (with some obvious adaptions):

$f(\text{meta}(\text{dis}(x))) = 2\times\text{term-size}(x) + 1$
$f(\text{meta}(\text{con}(x))) = 2\times\text{term-size}(x)$ □

So, with respect to characterizing termination, all issues seem solved. However, there is an angle. The idea of tuning techniques on recursion is entirely based on the intuition that,

for a terminating program and query, some data consumption takes place between every pair of descending calls to the same predicate. Recursion is only used to define problems in terms of more simple problems of the same type, using some well-founded recursive data structure. This, in turn, leads to the heuristic that for conditions tuned on recursion, straightforward level mappings, measuring the data consumption (e.g., the list-length or term-size of the input arguments), should be sufficient to prove termination. With the two approaches above, neither this intuition nor the heuristic is valid.

In the case of the "meta"-transformation, the transformed program has only one predicate. It completely hides the recursive structure of the original program. The decreases imposed by the generalized level mapping in the various conditions tuned on recursion of Section 3.2 all coincide with those imposed by recurrency and acceptability of Section 3.1: the level mapping must decrease for every body atom. As a result, more complex level mappings, such as the one in Example 4.12, are needed.

A similar conclusion holds for the first tentative solution. There, the adapted termination conditions impose a decrease on every intermediate call involved in a recursive loop. The intuition is that, in general, one can only expect data consumption after any full traversal of such a loop. Again, the solution requires more refined level mappings.

There are three ways in which automatic termination analysis can deal with these problems. The first is to apply transformations from mutually recursive to directly recursive programs that do not introduce new recursion within the transformed program, but only translate the given recursive structure into a directly recursive form. *Unfolding* is the main example. [90] proposes a solution based on it.

A second way out is to provide new termination conditions that are more refined than the ones sketched in our first tentative solution. One idea here is to select a minimal number of predicates involved in the mutual recursion and to impose decreases only for these predicates, thereby avoiding putting constraints on all intermediate predicates involved in the recursion. [116, 70, 26, 67], and [53] provide conditions of this type.

A last solution would be to improve the existing techniques for automatic generation of appropriate level mappings, given the program and queries of interest. These techniques could be extended to provide support for the generation of the level mappings needed in the termination proofs based on recurrency and acceptability. Although this seems feasible, we know of no work in this direction.

All mentioned works, either explicitly or implicitly, rely on the notion of a *predicate dependency graph* [90, 70, 67, 53] or the related notion of a *U-graph* [116], also referred to as a *specific graph* in [26]. The latter notion differs from the predicate dependency graph in the sense that unification is taken into account. More in particular, a U-graph contains a node for each *atom* in the program. It also contains two types of arcs: *clause arcs* and *unification arcs*. There is a clause arc from atom H to atom B if H is the head of a clause and B is a body atom of the same clause. There is a unification arc from atom B to atom H if B is a body atom and—after renaming—it unifies with the head, H, of another or the same clause. For the well-formed expressions example, the graphs are represented, respectively, in Figure 10 and Figure 11. In general, the U-graph is a more fine-grained representation for the recursive structure in a program.

In [90], Plümer suggests the use of algorithms from [66] and [3] as a basis for transforming mutually recursive programs into directly recursive ones by unfolding. These algorithms compute all *maximal strongly connected components* (MSCCs) and all *feedback nodes* of a predicate dependency graph. Here, a feedback node is a node contained in every cycle of the MSCC it belongs to. If every MSCC contains a feedback node, then the program can

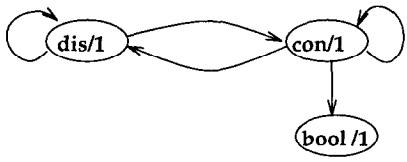

FIGURE 10. Predicate dependency graph for the program of Example 4.19.

be transformed into a directly recursive one by unfolding. The algorithms are linear-time.

For the program in Example 4.11, the condition above is not fulfilled. In fact, unfolding is insufficient to transform it into a directly recursive one. Still, it is hard to find sensible programs of this type. We only found parsing-type examples, similar to Example 4.11, in the literature. Therefore, the practical value of the result is very significant.

[90] also contains several other transformations. One allows us to transform *any* mutually recursive program into a directly recursive one with the same termination behavior, but it suffers from the same problems as the "meta"-transformation.

Hogger, in [70], also uses unfolding and—implicitly—the predicate dependency graph. Here, the aim is not to transform the program, but to formulate more refined termination conditions, tuned on recursion. The idea is to unfold atoms involved in mutual recursion to produce resultants of the unfolding (see [79]), in which the head and at least one body atom have the same predicate symbol. Let us refer to them as *directly recursive resultants*. The predicate dependency graph can be used to guide the unfolding to obtain such resolvents. Then, a decrease under some well-founded ordering, between the head and each body atom with the same predicate, is imposed on each directly recursive resultant.

Example 4.13. (parsing Boolean expressions)
For the program in Example 4.11, some directly recursive resultants for the dis/1 predicate are

$$\begin{aligned}
&\mathrm{dis}(b_1 \vee b_2) && \leftarrow \mathrm{con}(b_1),\mathrm{dis}(b_2).\\
&\mathrm{dis}((b_1 \& b_2) \vee b_3) && \leftarrow \mathrm{dis}(b_1),\mathrm{con}(b_2),\mathrm{dis}(b_3).\\
&\mathrm{dis}(b_1 \& b_2) && \leftarrow \mathrm{dis}(b_1),\mathrm{con}(b_2).
\end{aligned}$$

All of them satisfy a decrease between the dis atoms in the head and body of the resultants, using the well-founded ordering based on the term-size of their only argument. Note that this ordering is much simpler than the ones in Examples 4.11 and 4.12. □

The approach is intuitively appealing, but suffers from the problem that, in general, there may be infinitely many directly recursive resultants. It is unclear whether verifying the decrease for a finite number of them is always sufficient and, if so, how to select them.

The approaches in [26, 67], and [53] are conceptually similar to the one of Hogger. They inspect the dependencies in the program to detect all different ways in which a predicate may depend on itself. To do so, they collect all *elementary cycles* from the predicate dependency graph (in the case of [67] and [53]) or the U-graph (for [26]).

The associated directly recursive resultants are not explicitly constructed. Instead, rather technical termination conditions are formulated that ensure that

1. for any partial derivation corresponding to a full traversal of an elementary cycle, a given well-founded ordering on the atoms decreases;

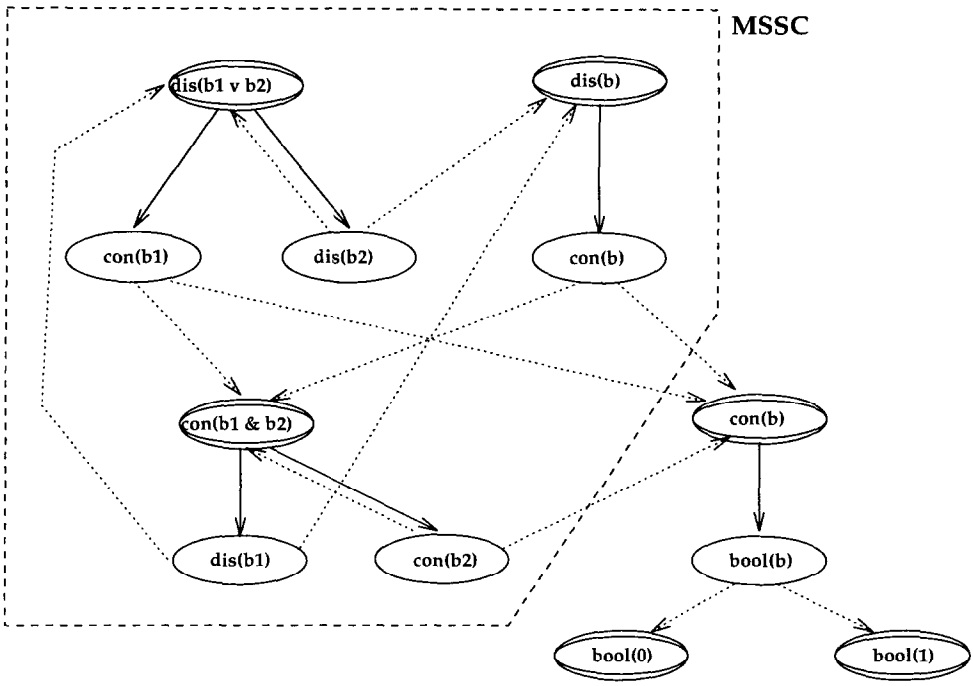

FIGURE 11. U-graph for the Boolean expressions example. Full arcs denote clause arcs, dotted arcs are unification arcs, double circled atoms are heads of clauses.

2. by intertwining full traversals of different elementary cycles (e.g., starting along one cycle and, intermediately, fully traversing another cycle, after which the traversal of the first one is completed), the decrease attached to the associated partial derivation is the composition of the decreases for the partial derivations associated to the individual elementary cycles.

By formulating their conditions to respect these two requirements, the approaches solve the problem of the potential existence of infinitely many different directly recursive resultants. On the other hand, by doing so, the technicality of the conditions is strongly increased. We do not present them in detail here, but refer to the papers instead. [111] contains a detailed discussion on the similarities and differences between [26] and [53].

Finally, we discuss the approach presented by Wang and Shyamasundar in [116] (see also [118, 117]). This work has introduced the notion of the U-graph and expresses termination conditions in terms of it. Again, the intuition is that one should impose a decrease of a well-founded measure with respect to all partial derivations associated to cycles in the graph. The key concept of the analysis is that of a *cycle cut of an MSCC* of the U-graph. A cycle cut, C, of an MSCC, S, is a set of clause arcs from S, such that every cycle in S contains at least one arc of C. Since a clause arc determines a pair of atoms, (H, B), such that H is the head of a clause and B a body atom of the same clause, termination conditions can easily be expressed in terms of a given cycle cut, S: there should be a decrease under the well-founded ordering between H and B, for every clause arc in S. Moreover, the order

should not increase for any of the arcs in $S \backslash C$.

Example 4.14. (parsing boolean expressions)
The U-graph in Figure 11 contains only one MSCC. One possible minimal cycle cut consists of the clause arcs:

$$(\mathrm{dis}(b_1 \vee b_2), \mathrm{dis}(b_2)), (\mathrm{con}(b_1 \& b_2), \mathrm{dis}(b_1)), (\mathrm{con}(b_1 \& b_2), \mathrm{con}(b_2)).$$

Again, the term-size norm applied to the unique argument of any of these atoms defines a well-founded ordering, satisfying the conditions of [116]. □

Compared with the previous approaches, the conceptual simplicity of the termination condition is an important advantage. The simplicity results from the fact that decreases are imposed locally, at the clause level. In the four previous approaches, decreases are imposed between atoms and descending atoms with the same predicate.

One can also relate the approach to the unfolding technique of Plümer via extensions of the latter for optimizing unfolding transformations. For instance, [29] contains an extension of the feedback node algorithm of [66] that, given an MSCC, S, computes a minimal set of nodes, N, of S, such that every cycle in S contains a node of N. The purpose of the algorithm is to control an unfolding transformation in order to produce a program with a minimal number of mutually recursive predicates.

5. CONCLUSIONS

Due to the way in which we decided to structure the paper, a few issues could not be classified within any of the sections. Some important ones are run-time termination techniques and cycle unification. They are discussed below. We end by stating some open problems.

5.1. Some Further Issues

5.1.1. Loop Detection. Up until now, the entire discussion has been devoted to compile-time termination analysis, with the possible exception of [107]. Important results concerning run-time detection and elimination of infinite loops are reported in [8, 22, 24, 34], and [108]. Most run-time approaches towards termination try to prune SLD-trees when some kind of repetition occurs, i.e., when the current resolvent is "sufficiently similar" to one of its ancestors. Examples of such *loop checks* are (see [22])

- the current *resolvent* is a *variant* of an ancestor,
- the current *resolvent* is an *instance* of an ancestor,
- the current *resultant* is a *variant* of an ancestor,
- the current *resultant* is an *instance* of an ancestor,
- the current (resolvent / resultant) *subsumes* a (variant/instance) of an ancestor.

Besides the above loop checks, several others were published before as well. Unfortunately, some of them are not always sound, in the sense that they may remove answers from an SLD-tree (see the discussion in [48, 49], and [94]). An example of an unsound loop check is the "variant of an atom check," which prunes a derivation at the first goal with a selected atom A, which is a variant of the selected atom A' of an earlier goal. To overcome this problem, loop checking was put on a firm theoretical basis by Bol et al. (see

[22] and [24]). The authors define important notions such as *soundness of a loop check* (no computed answer substitution to a goal is missed) and *completeness of a loop check* (all resulting derivations are finite), and they propose a number of concrete loop checks. For each of them, they prove an appropriate soundness result.

Besides formal correctness and precision, efficiency of the loop check is another important issue when trying to control termination at run time. The computational overhead must be reduced to the absolute minimum. This forms the main motivation for the work in [21] and [108]. The basic idea there is to compare only a restricted number of resolvents, without losing soundness.

Loop checking is also relevant in the context of controlling unfolding in partial deduction. Recent contributions can be found in [23, 102], and [36]. In this context, efficiency is, in general, not the major concern. Instead, emphasis lies on *completeness of the loop check* (unfolding is guided through a well-founded ordering or is forced to terminate by incorporating a depth-bound) and on *correctness and completeness of the transformation* (the residual program and the original program are equivalent).

An extension to loop checking is in some sense provided by *tabulation techniques* such as OLDT-resolution [106] and other, closely related approaches that were proposed in the literature (see, e.g., [104, 15], and [115]). All these techniques essentially tabulate answers for selected atoms. When a variant of such an atom is recursively called, execution is delayed and the selected atom is not further resolved; instead, all corresponding answers computed so far are looked up in the table, and execution is resumed by consecutively applying the corresponding answer substitutions to the delayed atom. This process is repeated for all subsequent computed answer substitutions that correspond to the atom. By doing so, a lot of potential loops are eliminated.

5.1.2. CYCLE UNIFICATION. As a second complement to the work discussed so far, we further elaborate on our comments on decidability and cycle unification. In the context of Term Rewriting Systems, [56] proves the undecidability of the halting problem for these systems by simulating a Turing machine by means of two rewrite rules. This result is improved in [50], where only one rewrite rule is needed. This has inspired work in LP to identify minimal classes of programs for which the problem is undecidable.

Research in this area has been focused on cycle unification. A cycle unification is defined by a Logic Program of the form

$$p(r_1, \ldots, r_n).$$
$$p(s_1, \ldots, s_n) \leftarrow p(t_1, \ldots, t_n).$$

augmented with a query $p(u_1, \ldots, u_n)$.

It has only one fact, one binary, directly recursive rule, and an atomic query, all with the same predicate symbol. It has been shown in [57] that minimal extensions of the expressivity of a cycle unification problem have the expressivity of a Turing machine. A main result from [57] is that termination of a cycle unification problem is decidable provided that both the query and the fact are *linear*. An atom is called linear if it does not contain the same variable twice. The nonterminating queries are characterized by means of a *weighted graph*, which captures the infinite sequence of unifications that the cycle unification produces.

In [20] and [119], related issues, such as 1) how many independent solutions does a cycle unification problem have, and 2) does there exist a unification algorithm which enumerates a minimal set of solutions for a cycle unification problem, are dealt with. [51] studies the problem on the basis of rational trees, and [52] provides an efficient decision procedure in the context of weighted graphs.

The decidability of cycle unification for nonlinear queries has finally been settled independently in [60] and [69]. It is proved that for nonlinear queries, the problem is undecidable, thereby solving a problem which has been open since 1973 [77], on the decidability of logic formulas involving only four atomic subformulas.

5.2. *Open Problems*

Of course, many issues remain, as yet, unsolved. In this section, we identify some key problems. We address the issues of

- existential termination and its relevance for the treatment of negation,
- improved inference of interargument relations,
- improved generation of well-founded orders,
- treatment of dynamic selection rules and of concurrent and constraint languages,
- comparability of different techniques.

5.2.1. EXISTENTIAL TERMINATION. Few works have addressed this problem. [65] studies the concept in the datalog context. [42] and [18] address the problem using inductive theorem proving.

The lack of work in this direction is not surprising: the problem is hard. It requires to prove that, following the order imposed by the selection rule, infinite derivations are preceded by a successful one. Clearly, if one makes any abstraction of the concrete data which are available in the considered queries, then proving that a derivation is successful (and not finitely failing) is difficult. Unifiability needs to be taken into account.

Nevertheless, more results on existential termination would be useful to provide more refined analysis techniques for normal programs. The reason is that a program terminates for a ground negative literal, $\neg A$ if and only if it existentially terminates for A. In the treatment of normal programs in Sections 2.2 and 3.1, this refinement is not taken into account. On the basis of a short initial study of the problem and of our knowledge of the technical problems involved in termination of cycle unification, we conjecture that *linearity* conditions, both on query and program, will be helpful to provide practical existential termination conditions. As an example, note that the only nonexistentially terminating query to append in Example 1.1, append($[x|y]$, Nil, y), is a nonlinear one.

5.2.2. IMPROVED INFERENCE OF INTERARGUMENT RELATIONS. In this context, we address the treatment of negation and open issues related to automation.

With respect to negation, the following problem turns up. Assuming that an interargument relation, R_p, has been inferred for a predicate p/n, what is the relation one should attach to a literal $\neg p(\bar{t})$? Intuitively, one would propose a relation based on the complement, $\mathbb{N}^n - R_p$. However, this is inadequate for two reasons. First, R_p is a safe approximation (superset) of the relation that holds between the sizes of the arguments of the computed answers for p. Thus, unless R_p is precise, the complement, $\mathbb{N}^n - R_p$, is not a superset for the relation imposed by the negation of p. Second, under the closed world assumption, negation as finite failure is more restrictive than classical negation. Therefore, even if a *precise* interargument relation R_p could be inferred, its complement might be an imprecise interargument relation for $\neg p(\bar{t})$ under the finite failure semantics.

Since selected negative literals are ground, one might question the relevance of inferring such interargument relations since their entire purpose is to deal with local variables. As

an example, a clause of the type

$$p(x) \leftarrow \neg q(x, y), p(y)$$

gives rise to a floundering derivation using the left-to-right selection rule.

However, as in Section 4.2, the point is that a negative literal might occur in combination with other intermediate body atoms. For instance, in the clause

$$p(x) \leftarrow \text{generate}(x, y), \neg\text{shorter\_or\_equal}(x, y), p(y)$$

the goal generate(x, y) might produce a ground list y, while shorter_or_equal(x, y) succeeds for pairs of lists (x, y), such that list-length$(x) \leq$ list-length(y). Here, the inference of the interargument relation for $\neg$ shorter_or_equal(x, y) — although easy in this case — is obviously crucial for the termination proof.

Another issue related to interargument relations is the automation, evaluation, and integration into a termination analysis system of the more refined inference techniques presented in Section 4.2.4 and in the conclusions of Section 4.2. The increased expressivity and the high precision that could, in principle, be obtained using the approaches of [32] and [109] are desirable. Although [46] sketches a possible automation using abstract interpretation, as far as we know, no successful experiments were carried out to fill in the details and integrate the approaches in a termination analysis. The required mathematical machinery seems heavy. Also, deciding upon some tradeoffs involved in the requirements of precision and efficiency seems to need considerable work.

5.2.3. Generation of Useful Well-Founded Orders. With respect to the selection of appropriate well-founded orders, a number of issues are unresolved.

First, given a norm, the method of Sohn and Van Gelder sketched in Section 4.1 allows us to generate an appropriate level mapping, defined as a particular linear combination of the norms of certain argument positions. In some cases, however, level mappings defined in terms of more general functions of the used norm (e.g., maximum or minimum) are known to be more useful. Example 3.2 illustrates the point. Whether such level mappings could also be generated in a systematic way is unclear.

A related issue is providing automatic support for the generation of the level mappings needed in the notions of recurrency and acceptability (or semi-recurrency and semi-acceptability). As argued in Section 4.3, with these concepts, often some seemingly unnatural offsets or coefficients turn up in the level mapping. These can clearly be related to syntactic properties of the program, and we are convinced that their generation could be automated to a reasonable extent. In particular, this could provide an interesting alternative to the automatic treatment of mutual recursion, based on semi-recurrency and semi-acceptability. The gain would be that the complexity involved in treating mutually recursive programs would not be noticeable in the termination conditions, but in the level mapping generation instead.

A next issue is the generation of norms. [54], discussed in Section 4.1, presents a first contribution. A set of potentially useful norms is generated on the basis of type inference. Further delimiting the search for an appropriate norm is challenging. An approach based on linear programming, similar to that of Sohn and Van Gelder for level mappings, seems most promising. Symbolic norms could be introduced and termination conditions could be formulated as constraints on the symbolic coefficients in the norm.

Nevertheless, the problem seems more difficult than the generation of level mappings. The point is that interargument relations are needed to formulate termination conditions

and norms are needed to infer the interargument relations. Furthermore, inferring the interargument relations on the basis of a symbolic norm seems unfeasible since it involves global analysis. Thus, the problem is that concrete norms seem necessary to compute interargument relations, while interargument relations are needed to compute concrete values for the coefficients in a symbolic norm. A breakthrough is needed here.

A next issue is whether orderings based on the syntactic subterm ordering could be sufficient to deal with local variables in practice. As far as we know, the problem has not been studied in any depth so far, while it is of particular relevance to evaluate the practicality of elegant approaches, such as [33].

A final and related issue is the study of more general well-founded orders than those based on norms and level mappings. In the definition of a norm (or level mapping), one could replace $\mathbb{N}$ by any well-founded ordered set. The gain is that new types of orders, such as lexicographical orders over $\mathbb{N}^n$, can be incorporated. It is well known that, in some cases, termination of a program cannot be proved without resorting to such more refined orderings. The Ackerman function is a typical example, in which any natural termination proof requires a lexicographical ordering over $\mathbb{N}^2$. Also, in the study of coroutining executions of programs, such orderings naturally show up (we return to coroutining in the next subsection). In [82], the use of more general well-founded orders is studied in the context of control of unfolding in partial deduction. A similar study in the context of termination analysis would be desirable, in particular since it would allow us to further clarify the relations with the work on termination of Term Rewriting.

5.2.4. Treatment of Dynamic Selection Rules and of Concurrent and Constraint Languages. The left-to-right selection rule has been standard in most logic programming languages in the past. The current trend is to support more flexible, dynamic selection rules. Many systems support delay declarations, concurrency, or even both. We refer to [88] for a brief overview of such systems. Termination analysis for derivations following a dynamic computation rule requires specialized techniques. In particular, analyzing termination of coroutining procedures is an important open problem.

Two recent contributions to this problem are [93] and [97]. These works adapt previously developed techniques for analyzing sequential logic programs, proposed in, respectively, [90] and [96], to GHC-programs. Both take a transformational approach. [93] simulates GHC-computations by means of sequential SLD-derivations, and then applies the termination analysis of [90]. [97] adapts the transformation scheme to Term Rewrite Systems of [96] to the GHC-context. The latter approach seems to be more naturally applicable to the GHC- than to the SLD-context. One of the reasons is that unification is already restricted to matching, which allows us to avoid the earlier requirement of having a well-moded* query. The main limitation of these approaches is that they are both incapable of adequately treating coroutining in its full generality. The key point seems to be that coroutining, in general, involves cyclic producer–consumer relations. Since in both techniques the termination analysis applied to the transformed program (respectively, a sequential logic program and a TRS) requires some form of producer–consumer acyclicity of the program, general coroutines are beyond their scope.

In [88], L. Naish indicates some directions for more generally applicable solutions. It is emphasized that a combination of mode and type information is essential to conduct a sufficiently refined analysis. The purpose of such information is to allow a precise description of the dynamic data flow. Contributions are made on two different levels. A first result deals with the termination of derivations, under a dynamic selection rule, for a query which is a conjunction of atoms, such that the derivations of each separate atom terminate and

such that the procedures defining the atoms have no recursive interdependencies. Given an acyclicity condition on the producer–consumer relation in the conjunction (and some additional conditions on the dynamic selection rule), it is proved that the termination of the individual atoms is inherited by the conjunction.

A more speculative contribution is on the treatment of general coroutining, involving recursive dependencies between the atoms in the conjunction. Here, performing *speculative output bindings* is identified as the main cause for potential divergence of the computation. A typical example of speculative output bindings is presented in the permute example in Section 3.3. In this example, following a nonstandard selection rule, output bindings for delete atoms are generated before it is known that the calls to these atoms may succeed. Due to these bindings, the recursive inference is allowed to proceed while, in fact, full evaluation of an ancestor call would establish failure.

Termination analysis could detect such diverging computations by associating a lexicographical well-founded ordering with conjunctions of atoms. Some data structures involved in a coroutining computation could then be allowed to increase in size, as long as a data structure with a higher priority (possibly occurring in another atom) decreases. This is completely in the spirit of our comments in Section 5.2.3. Further work on the issue seems essential.

Constraint Logic Programming provides another important new class of LP languages. Here, the new issues in termination analysis are the interaction between the external solvers and SLDNF-resolution and—especially for finite domain CLP—the dynamic selection rules used to enforce first-fail principles on the activation order for the constraints.

[84] provides an initial study of the topic. It generalizes many of the concepts and techniques developed for LP within the CLP context. A basic notion which is introduced is that of an approximation between two CLP languages. Such approximations are functions that map each program of a first CLP language to an associated program of the second language, with the fundamental property that any model of the associated program corresponds—through composition with the approximation itself—to a model of the initial program. The inverse of the approximation preserves the termination property with respect to the different procedural semantics of the two languages. The latter allows us to study the termination behavior of an approximated program, and to use the results to infer termination properties of an original program. What is gained here is that useful approximations will focus only on those aspects of the given program that are relevant for the termination behavior. For instance, on the size of the terms occurring in the program, or on booleans expressing decreases between the sizes of terms in the head of rules and sizes of corresponding terms in the body. As such, the notion of an approximation is related to the notion of a norm. The main relevance of this contribution is that it illustrates how CLP termination is useful in providing a more elegant treatment of termination analysis. Through various levels of constraint languages, the termination problem of a lower level language can be modeled by means of a CLP program at a higher level. Since LP is a specific instance of a CLP language, this also provides a new and elegant view on LP termination analysis.

Another important contribution is that the outcome of the analysis is a constraint, such that all constraint atoms satisfying this constraint terminate under the considered selection rule. This is similar, but more general, to the flexibility obtained in inferring termination for all queries that are bounded with respect to a level mapping (see Section 4.1).

5.3. *Comparability of the Different Techniques*

More work is also needed on evaluating and comparing the power of different automated techniques. How do we identify the class of programs for which the termination conditions of a certain technique are sufficient? In current work, these classes are usually indicated by means of some examples. Whether better classifications are feasible is unclear.

The problem is related to the undecidability issue. Providing decidable sufficient conditions necessarily involves some ad hoc decisions based on heuristics. Often, however, even these heuristics are not made explicit by the authors. They are hidden in some restrictions related to technicalities of the approach. The impact of such restrictions on the applicability may be unclear, while they definitely obscure the relation to other approaches, imposing different restrictions.

Another reason why comparison is difficult is that most techniques propose their own specialized method for global program analysis. The disadvantage is that issues related to the structure and the design of the global analysis are mixed with issues related to the specific application: termination analysis. On the other hand, specialized analysis is often more efficient. Still, we believe that the use of generic frameworks for global analysis of Logic Programs, e.g., based on abstract interpretation, would allow us to significantly improve the comparability of techniques. Such frameworks are widely available (see, e.g., [46]).

We wish to thank K. Apt, E. Bevers, R. Bol, A. Bossi, D. Boulanger, M. Bruynooghe, N. Cocco, B. Demoen, P. Devienne, M. Fabris, C. Hogger, S. Hölldobler, A. Mariën, B. Martens, F. Mesnard, D. Pedreschi, A. Pettorossi, L. Plümer, K. Rao, and K. Verschaetse for interesting discussions on the topic. Also thanks to E. Bevers, M. Bruynooghe, M. Leuschel, and B. Martens for proofreading drafts of the paper, to K. Verschaetse for allowing us to adapt a few paragraphs from his Ph.D., and to C. Hogger for inspiring the title. Finally, we wish to thank anonymous referees for valuable comments. At this point, we justify the title by presenting an extensive survey on: Termination of Logic Programs: the Never-Ending Story (see first page).

REFERENCES

1. Afrati, F., Papadimitriou, C., Papageorgiou, G., Roussou, A. R., Sagiv, Y., and Ullman, J. D., On the Convergence of Query Evaluation, *Journal of Computer and System Sciences* 38(2):341–359 (1989).
2. Aguzzi, G., and Modigliani, U., Proving Termination of Logic Programs by Transforming them into Equivalent Term Rewrite Systems, in: *Proceedings of Foundations of Software Technology And Theoretical Computer Science*, 1993.
3. Aho, A., Hopcroft, J., and Ullman, J., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.
4. Apt, K. R., Marchiori, E., and Palamidessi, C., A Theory of First-Order Built-In's of Prolog, in: *Proceedings of Algebraic and Logic Programming 92*, 1992.
5. Apt, K. R., and Pellegrini, A., Why the Occur-Check is not a Problem, in: M. Bruynooghe and M. Wirsing (eds.), *PLILP92*, Leuven, Belgium, 1992, pp. 69–86, Springer-Verlag, LNCS 631.
6. Apt, K. R., Logic Programming, in: J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science, Vol. B*, Elsevier Science Publishers, 1990.
7. Apt, K. R., and Bezem, M., Acyclic Programs, *New Generation Computing* 9:335–363 (1991).
8. Apt, K. R., Bol, R. N., and Klop, J. W., On the Safe Termination of Prolog Programs, in: *Proceedings ICLP'89*, Lisbon, June 1989, pp. 353–368, MIT Press.

9. Apt, K. R., and Marchiori, E., Reasoning About Prolog Programs: From Modes Through Types to Assertions, Technical Report CS-R9358, CWI, Aug. 1993.
10. Apt, K. R., Marchiori, E., and Palamidessi, C., A Theory of First-Order Built-In's of Prolog, in: H. Kirchner and G. Levi (eds.), *Algebraic and Logic Programming, Proceedings of the Third International Conference*, Berlin, 1992, pp. 69–83, Springer-Verlag, LNCS 632.
11. Apt, K. R., and Pedreschi, D., Studies in Pure Prolog: Termination, in: *Proceedings Esprit Symposium on Computational Logic*, Brussels, November 1990, pp. 150–176, Springer-Verlag.
12. Apt, K. R., and Pedreschi, D., Proving Termination of General Prolog Programs, in *Proceedings International Conference on Theoretical Aspects of Computer Science*, Sendai, Japan, 1991.
13. Apt, K. R., and Pedreschi, D., Modular Termination Proofs for Logic and Pure Prolog Programs, Technical Report 6/93, Dipartimento di Informatica, Universita di Pisa, 1993.
14. Balbiani, P., The Finiteness of Logic Programming Derivations, in: H. Kirchner and G. Levi (eds.), *Algebraic and Logic Programming, Proceedings of the Third International Conference*, Berlin, 1992, pp. 403–419, Springer-Verlag, LNCS 632.
15. Bancilhon, F., and Ramakrishnan, R., An Amateur's Introduction to Recursive Query Processing Strategies, in: *Proceedings SIGMOD86*, 1986.
16. Barbuti, R., Codisch, M., Giacobazzi, R., and Maher, M., Oracle Sematics for Prolog, in: *Proc. Third Algebraic and Logic Programming International Conference*, pp. 100–114, Springer-Verlag, LNCS 632. To appear in *Journal of Logic Programming*.
17. Barbuti, R., Giacobazzi, R., and Levi, G., A General Framework for Semantics-Based Bottom-Up Abstract Interpretation of Logic Programs, *ACM Transactions on Programming Languages and Systems* 15(1):133–181 (1991).
18. Baudinet, M., Proving Termination Properties of Prolog Programs: A Semantic Approach, *Journal of Logic Programming* 14:1–29 (1992).
19. Bezem, M., Characterizing Termination of Logic Programs with Level Mappings, *Journal of Logic Programming* 15(1 & 2):79–98 (1992).
20. Bibel, W., Hölldobler, S., and Würtz, J., Cycle Unification, in: D. Kapur (ed.), *CADE*, 1992, pp. 94–108, Springer, LNCS 607.
21. Bol, R. N., Towards More Efficient Loop Checks, in: *Proceedings NACLP'90*, 1990, pp. 465–479.
22. Bol, R. N., Loop Checking in Logic Programming, Ph.D. thesis, University of Amsterdam, Oct. 1991.
23. Bol, R. N., Loop Checking in Partial Deduction, *Journal of Logic Programming* 16(1 & 2):25–46 (1993).
24. Bol, R. N., Apt, K. R., and Klop, J. W., An Analysis of Loop Checking Mechanisms for Logic Programs, *Theoretical Computer Science* 86(1):35–79 (Aug. 1991).
25. Boolos G., and Sambin, G., Provability: The Emergence of a Mathematical Modality, *Studia Logica* 1–23 (1991).
26. Bossi, A., Cocco, N., and Fabris, M., Norms on Terms and Their Use in Proving Universal Termination of a Logic Program, Technical Report 4/29, CNR, Department of Mathematics, University of Padova, Mar. 1991. To appear in *Theoretical Computer Science*.
27. Bossi, A., Cocco, N., and Fabris, M., Proving Termination of Logic Programs by Exploiting Term Properties, in: *Proc. CCPSD-TAPSOFT'91*, 1991, pp. 153–180, Springer-Verlag, LNCS 494.
28. Bossi, A., Cocco, N., and Fabris, M., Typed Norms, in: B. Krieg-Brueckner (ed.), *Proc. ESOP'92*, 1992, pp. 73–92, Springer-Verlag, LNCS 582.
29. Boulanger, D., and Bruynooghe, M., Deriving Fold/Unfold Transformations of Logic Programs Using Extended Oldt-Based Abstract Interpretation, *Journal of Symbolic Computation* (1993).
30. Brodsky, A., and Sagiv, Y., Inference of Monotonicity Constraints in Datalog Programs,

in: *Eighth ACM Symposium on Principles of Database Systems*, 1989, pp. 190–199.
31. Brodsky, A., and Sagiv, Y., On Termination of Datalog Programs, in: *First International Conference on Deductive and Object Oriented Databases*, Kyoto, Japan, 1989, pp. 95–112.
32. Brodsky A., and Sagiv, Y., Inference of Inequality Constraints in Logic Programs, in: *Proceedings of the 10th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Denver, CO, 1991, pp. 227–240.
33. Bronsard, F., Lakshman, T. K., and Reddy, U. S., A Framework of Directionality for Proving Termination of Logic Programs. in: K. Apt (ed.), *Proc. JICSLP '92*, 1992, pp. 321–335, MIT Press.
34. Brough, D. R., and Walker, A., Some Practical Properties of Logic Programming Interpreters, in: *Proceedings FGCS'84*, 1984, pp. 149–156.
35. Bruynooghe, M., A Practical Framework for the Abstract Interpretation of Logic Programs, *Journal Logic Programming* 10(2):91–124 (1991).
36. Bruynooghe, M., De Schreye, D., and Martens, B., A General Criterion for Avoiding Infinite Unfolding During Partial Deduction of Logic Programs, *New Generation Computing* 11(1):47–79 (1992).
37. Cartwright, R., Recursive Programs as Definitions in First Order Logic, *SIAM J. Comput.* 13(2):374–408 (1984).
38. Cartwright, R., and Mc Carthy, J., First Order Programming Logic, in: *Sixth ACM Symposium on Principles of Programming Languages*, San Antonio, TX, 1979, pp. 68–80.
39. Cartwright, R., and Mc Carthy, J., Recursive Programs as Functions in a First Order Theory, in: *Proceedings of the 1978 International Conference on Mathematical Studies of Information Processing*, Kyoto, Japan, 1979.
40. Cavedon, L., Continuity, Consistency, and Completeness Properties for Logic Programs, in: *Proceedings ICLP'89*, June 1989, pp. 571–584.
41. Chan, D., Constructive Negation Based on the Completed Database, in: R. A. Kowalski and K. A. Bowen (eds.), *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, Seattle, WA, 1988, pp. 111–125, ALP, IEEE, MIT Press.
42. Clark, K. L., and Tärnlund, S.-Å., A First Order Theory of Data and Programs, in: B. Gilchrist (ed.), *Information Processing 77, Proceedings of the IFIP Congress 77*, Toronto, 1977, pp. 939–944.
43. Colmerauer, A., An Introduction to PROLOGIII, *Communications of the ACM* 30(7):69–96 (1990).
44. Colussi, L., and Marchiori, E., Proving Correctness of Logic Programs Using Axiomatic Semantics, in: K. Furukawa (ed.), *Proceedings ICLP91*, 1991, pp. 629–642, MIT Press.
45. Cook, J., and Gallagher, J., A Transformation System for Definite Programs Based on Termination Analysis, Technical Report CSTR-92-32, Univ. of Bristol, Nov. 1992.
46. Cousot, P., and Cousot, R., Abstract Interpretation and Application to Logic Programming, *Journal of Logic Programming* 13(2 & 3):103–180 (1992).
47. Cousot, P., and Halbwachs, N., Automatic Discovery of Linear Restraints Among Variables of a Program, in: *Proceedings 5th ACM Symposium on Principles of Programming Languages*, 1978, pp. 84–96.
48. Covington, M., Eliminating Unwanted Loops in Prolog, *Sigplan Notices* 20(1):20–26 (Jan. 1985).
49. Covington, M., A Further Note on Looping in Prolog, *Sigplan Notices* 20(8):28–31 (Aug. 1985).
50. Dauchet, M., Simulation of Turing Machines by a Left-Linear Rewrite Rule, in: *Proceedings of the 3rd International Conference on Rewriting Techniques and Applications*, Chapel Hill, NC, 1989.
51. De Schreye, D., Bruynooghe, M., and Verschaetse, K., On the Existence of Nonterminating Queries for a Restricted Class of Prolog-Clauses, *Artificial Intelligence* 41:237–248

(1989).
52. De Schreye, D., Verschaetse, K., and Bruynooghe, M., A Practical Technique for Detecting Nonterminating Queries for a Restricted Class of Horn Clauses, Using Directed, Weighted Graphs, in: *Proceedings ICLP'90*, Jerusalem, June 1990, pp. 649–663, MIT Press.
53. De Schreye, D., Verschaetse, K., and Bruynooghe, M., A Framework for Analysing the Termination of Definite Logic Programs with Respect to Call Patterns, in: *Proc. FGCS'92, International Conference on Fifth Generation Computer Systems*, ICOT Tokyo, 1992, pp. 481–488. ICOT.
54. Decorte, S., De Schreye, D., and Fabris, M., Automatic Inference of Norms: A Missing Link in Automatic Termination Analysis, in: D. Miller (ed.), *Proceedings ILPS'93*, Vancouver, Canada, 1993, pp. 420–436.
55. Dembinski, P., and Maluszynski, J., AND-Parallelism with Intelligent Backtracking for Annotated Logic Programs. in: *Proceedings of the International Symposium on Logic Programming*, Boston, 1985, pp. 29–38.
56. Dershowitz, N., Termination of Rewriting, *Symbolic Computation* 3(1 & 2):69–116 (1987).
57. Devienne, P., Weighted Graphs: A Tool for Studying the Halting Problem and Time Complexity in Term Rewriting Systems and Logic Programming, *Theoretical Computer Science* 75(1 & 2):157–215 (1990).
58. Devienne, P., and Lebègue, P., Weighted Gaphs, A Tool for Logic Programming, in: *11th Colloquium on Trees in Algebra and Programming*, Nice, 1986, pp. 100–111
59. Devienne, P., Lebègue, P., and Dauchet, M., Weighted Systems of Equations, Technical Report IT 188, Laboratoire d'Informatique Fondamentalle de Lille, France, May 1990.
60. Devienne, P., Lebègue, P., and Routier, J. C., Halting Problem of One Binary Horn Clause Is Undecidable, in: *Proceedings of STACS'93*, Würzburg, 1993, Springer-Verlag.
61. Deville, Y., *Logic Programming: Systematic Program Development*, Addison-Wesley, 1990.
62. Drabent, W., and Maluszynski, J., Inductive Assertion Method for Logic Programs, *Theoretical Computer Science* 59:133–155 (1988).
63. Falaschi, M., Levi, G., Martelli, M., and Palamidessi, C., Declarative Modeling of the Operational Behaviour of Logic Languages. *Theoretical Computer Science* 69(3):289–318 (1989).
64. Floyd, R. W., Assigning Meanings to Programs, in: *Proceedings Symp. in Applied Math.*, vol. 19, Providence, RI, 1967, pp. 19–32, Amer. Math. Soc.
65. Franchez, N., Grumberg, O., Katz, S., and Pnueli, A., Proving Termination of Prolog Programs, in: R. Parikh (ed.), *Logics of Programs*, Springer-Verlag, 1985, pp. 89–105,
66. Garey, M., and Tarjan, R., A Linear-Time Algorithm for Finding All Feedback Vertices, in: *Information Processing Letters* 7:274–276 (1978).
67. Gröger, G., and Plümer, L., Handling of Mutual Recursion in Automatic Termination Proofs for Logic Programs, in: K. Apt (ed.), *Proceedings of the Joint International Conference and Symposium on Logic Programming*, MIT Press, 1992, pp. 336–350.
68. Hanks, S., and McDermott, D., Nonmonotonic Logic and Temporal Projection, *Artificial Intelligence* 33:379–412 (1987).
69. Hanschke, P., and Würtz, J., Satisfiability of the Smallest Binary Program, in: *Information Processing Letters* 45(9):237–241 (Apr. 1993).
70. Hogger, C. J., *Essentials of Logic Programming*, Oxford University Press, 1990.
71. Kapur, D., and Zhang, H., An Overview of Rewrite Rule Laboratory (RRL), in: *Proceedings of Rewrite Techniques and Applications Conference*, vol. LNCS 355, Springer-Verlag, 1989, pp. 559–563.
72. Karr, M., Affine Relationships Among Variables of a Program, *Acta Informatica* 6:133–151 (1976).
73. Kifer, M., Ramakrishnan, R., and Silberschatz, A., An Axiomatic Approach to Deciding Query Safety in Deductive Databases. in: *Proceedings ACM Symposium on Management*

of Data, Chicago, 1988, pp. 154–163.
74. Kowalski, R. A., Algorithm = Logic + Control, *Communications of the ACM* 22:424–431 (1979).
75. Krishnamurthy, R., Ramakrishnan, R., and Shmueli, O., A Framework for Testing Safety and Effective Computability of Extended Datalog. in: *Proceedings 7th ACM Symposium on Principles of Database Systems*, Austin, TX, 1988, pp. 52–60.
76. Lescanne, P., Computer Experiments With the Reve Term Rewriting Systems Generator, in: *Proceedings 10th ACM Symp. on Principles of Programming Languages POPL'83*, 1983.
77. Lewis, H., and Goldfarb, W. D., The Decision Problem for Formulas with a Small Number of Atomic Subformulas, *Journal of Symbolic Logic* 38(3):471–480 (1973).
78. Lloyd, J. W., *Foundations of Logic Programming*, Springer-Verlag, 1987.
79. Lloyd, J. W., and Shepherdson, J. C., Partial Evaluation in Logic Programming, *Journal of Logic Programming* 11(3 & 4):217–242 (Oct./Nov. 1991).
80. Manna, Z., and Ness, S., On the Termination of Markov Algorithms, in: *Proceedings 3rd Hawaii Int. Conf. on Syst. Sci.*, Honolulu, HI, 1970.
81. Marchiori, E., Proving Run-Time Properties of General Programs w.r.t. Constructive Negation, Technical Report CS-R9245, CWI, Amsterdam, Nov. 1992.
82. Martens, B., and De Schreye, D., Advanced Techniques in Finite Unfolding, Technical Report CW180, K.U. Leuven, Belgium, Oct. 1993.
83. Mellish, C. S., Some Global Optimizations for a Prolog Compiler, *Journal of Logic Programming* 2(1) (Apr. 1985).
84. Mesnard, F., Etude de la Terminaison des Programmes Logiques avec Constraintes aux Moyens d'Approximations, Ph.D. thesis, Paris VI, 1993.
85. Mesnard, F., and Ganascia, J., CLP(Q) for Proving Interargument Relations, in: A. Pettorossi (ed.), *Proceedings META'92*, Uppsala, Sweden, June 1992, pp. 309–320, Springer-Verlag.
86. Naish, L., Automating Control for Logic Programs, *Journal of Logic Programming* 2(3) (Oct. 1985).
87. Naish, L., *Negation and Control in Prolog*, LNAI 238, Springer-Verlag, 1986.
88. Naish, L., Coroutining and the Construction of Terminating Logic Programs, Technical Report 92/5, University of Melbourne, Australia, 1992.
89. Nilsson, N. J., *Principles of artificial intelligence*, Los Altos, CA, Morgan Kaufmann, 1980.
90. Plümer, L., *Termination Proofs for Logic Programs*, number 446 in LNAI, Springer-Verlag, 1990.
91. Plümer, L., Termination Proofs for Logic Programs Based on Predicate Inequalities, in: *Proceedings ICLP'90*, Jerusalem, June 1990, pp. 634–648, MIT Press.
92. Plümer, L., Automatic Termination Proofs for Prolog Programs Operating on Nonground Terms, in: *Proceedings ILPS'91*, San Diego, October 1991, pp. 503–517, MIT Press.
93. Plümer, L., Automatic Verification of GHC-Programs: Termination, in: *Proceedings FGCS'92*, Tokyo, 1992.
94. Poole, D., and Goebel, R., On Eliminating Loops in Prolog, *Sigplan Notices* 20(8):38–40 (Aug. 1985).
95. Ramakrishnan, R., Bancilhon, F., and Silberschatz, A., Safety of Recursive Horn Clauses with Infinite Relations, in: *Proceedings 6th Symposium on Principles of Database Systems*, ACM Press, 1987, pp. 328–339.
96. Krishna Rao, M. R. K., Kapur, D., and Shyamasundar, R. K., A Transformational Methodology for Proving Termination of Logic Programs, in: *Proceedings CSL'91, LNCS 626*, Springer, 1991.
97. Krishna Rao, M. R. K., Kapur, D., and Shyamasundar, R. K., Proving Termination of GHC Programs, in: *proceedings ICLP93*, Boedapest, 1993, pp. 720–736.
98. Krishna Rao, M. R. K., Pandya, P. K., and Shyamasundar, R. K., Verification Tools in

the Development of Provably Correct Compilers, in: *Proceedings 5th Symp. on Formal Methods Europe, FME'93*, 1993.

99. Rosenblueth, D., Chart Parsers as Proof Procedures for Fixed-Mode Logic Programs, in: *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT, Japan, 1992, pp. 1125–1132, Association for Computing Machinery.
100. Sagiv, Y., A Termination Test for Logic Programs, in: *Proceedings ILPS'91*, San Diego, CA, 1991, pp. 518–532, MIT Press.
101. Sagiv, Y., and Vardi, M. Y., Safety of Datalog Queries over Infinite Databases, in: *Proceedings ACM SIGACT-SIGART-SIGMOD Symposium on Principles of Database Systems*, Philadelphia, PA, 1989, pp. 160–171, Academic Press.
102. Sahlin, D., The Mixtus Approach to Automatic Partial Evaluation of Full Prolog, in: S. Debray and M. Hermenegildo (eds.), *Proceedings NACLP'90*, 1990, pp. 377–398.
103. Shyamasundar, R. K., Krishna Rao, M. R. K., and Kapur, D., Rewriting Concepts in the Study of Termination of Logic Programs, in: *Proc. ALPUK'92*, London, Apr. 1992.
104. Smith, D. E., Genesereth, M. R., and Ginsberg, M. L., Controlling Recursive Inference, *Artificial Intelligence* 30:343–389 (1986).
105. Sohn, K., and Van Gelder, A., Termination Detection in Logic Programs Using Argument Sizes, in: *Proceedings 10th Symposium on Principles of Database Systems*, ACM Press, May 1991, pp. 216–226.
106. Tamaki, H., and Sato, T., OLD Resolution with Tabulation, in: *Proceedings ICLP'86*, Lecture Notes in Computer Science 225, Springer Verlag, 1986, pp. 84–98.
107. Ullman, J. D., and Van Gelder, A., Efficient Tests for Top-Down Termination of Logical Rules, *Journal ACM* 35(2):345–373 (Apr. 1988).
108. Van Gelder, A., Efficient Loop Detection in Prolog Using the Tortoise-and-Hare Technique, *Journal of Logic Programming* 4:23–31 (1987).
109. Van Gelder, A., Deriving Constraints Among Argument Sizes in Logic Programs, in: *Proc. PODS91*, ACM Press, April 1990, pp. 47–60.
110. Vasak, T., and Potter, J., Characterisation of Terminating Logic Programs, in: *Proceedings 1986 Symposium on Logic Programming*, Salt Lake City, UT, 1986, pp. 140–147.
111. Verschaetse, K., Static Termination Analysis for Definite Horn Clause Programs, Ph.D. thesis, Dept. Computer Science, K.U.Leuven, 1992.
112. Verschaetse, K., and De Schreye, D., Deriving Termination Proofs for Logic Programs, Using Abstract Procedures, in: *Proc. ICLP'91*, Paris, June 1991, pp. 301–315, MIT Press.
113. Verschaetse, K., and De Schreye, D., Derivation of Linear Size Relations by Abstract Interpretation, in: M. Bruynooghe and M. Wirsing (eds.), *Proceedings PLILP'92, 4th International Symposium on Programming Language Implementation and Logic Programming*, LNCS 631, Springer-Verlag, 1992, pp. 296–310.
114. Verschaetse, K., Decorte, S., and De Schreye, D., Automatic Termination Analysis, in: *Proc. LOPSTR'92, LNCS*, Springer-Verlag, 1993.
115. Vieille, L., Recursive Query Processing: The Power of Logic, *Theoretical Computer Science* 69(1):1–53 (1989).
116. Wang, B., and Shyamasundar, R. K., Towards a Characterization of Termination of Logic Programs, in: *Proc. PLILP'90*, number 456 in LNCS, Linköping, Sweden, Aug. 1990, pp. 204–221, Springer-Verlag.
117. Wang, B., and Shyamasundar, R. K., Methodology for Proving the Termination of Logic Programs, in: *Proceedings STACS'91*, number 480 in LNCS, Hamburg, Germany, 1991, pp. 214–227.
118. Wang, B., and Shyamasundar, R. K., Proving Termination of Logic Programs, in: R. Narasimhan (ed.), *Perspective in Theoretical Computer Science, Commemorative Volume*, World Scientific Publishers, 1989, pp. 380–397.
119. Würtz, J., Unifying Cycles, in: B. Neumann (ed.), *ECAI*, 1992, pp. 60–64.